ODÉON:
An Object-Oriented Data Model and its Integration in the Oberon System

Jacques Supcik

Diss. ETH No 13161

ODÉON

An Object-Oriented Data Model
and its Integration in the Oberon System

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH
(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
Jacques Supcik
Dipl. Informatik-Ing. ETH
born August 30, 1967
citizen of Fribourg, Switzerland

accepted on the recommendation of
Prof. Dr. Niklaus Wirth, examiner
Prof. Dr. Moira C. Norrie, co-examiner

1999

A mes parents

Acknowledgements

This work would not have been possible without the help and encouragement of many people. I am most indebted to Prof. N. Wirth for his continuous support and his insistence on clarity, precision and simplicity. Without his constructive criticism, this project would not have achieved such favorable results. I also would like to thank Prof. M. Norrie for having made me discover the fantastic world of object oriented database systems and for acting as my co-examiner. Her deep knowledge in the domain of persistent stores, persistent languages and object oriented database systems was always a priceless source of inspiration for me.

It is a pleasure to mention my colleagues of the Institute for Computer Systems and of the Institute for Information Systems. They established an agreeable and inspiring environment and enjoyable working atmosphere.

Régis Crelier helped me a lot for the implementation of HP-Oberon and was always ready to discuss solutions to problems encountered.

Marco Sanvido proofread parts of the manuscript.

Stefan Ludwig gave me valuable advice, especially for the garbage collector and for the implementation of the finalizer.

Alain Würlger and Adrian Kobler have implemented other variants of OMS and it was always interesting to share ideas with them.

Thank you to Marc Brandis and Josef Templ for their fruitful and exciting discussions during coffee breaks.

Fiona Blackburn carefully proofread most of the manuscript and corrected many language mistakes and inelegance.

I am also thankful to all students who did term- and diploma-works around the Odéon project. They helped me a lot in experimenting with new ideas.

I gratefully acknowledge all the support I have received over the years from my colleagues at ETH, who are just too numerous to be mentioned in

detail.

The implementation of Oberon on HP has been supported by Hewlett-Packard Switzerland by their generous lending of a powerful workstation.

Contents

I Ideas and Concepts **1**

1 Introduction **3**

1.1 The Current Situation 3
1.2 Why Odéon? 5
1.3 Contributions 6
1.4 Structure of the Thesis 6

2 Background **9**

2.1 The Persistent Store 9
2.2 The Object Oriented Data Model 11

3 The Data Model **17**

3.1 What is an Object? 18
3.2 Classification by Collections 20
3.3 Typing by Classification 23
3.4 Inheritance and the ISA Relation 25
3.5 Associations 27
3.6 The Root Objects 29

4 Evolution **31**

4.1 Object Evolution 31
4.2 Schema Evolution 34

5 Query and Algebra **37**

5.1 Query Processing 37
5.2 The Algebra 39
5.2.1 Operations on Collections 39
5.2.2 Operation on Associations 42

6 Constraints **47**
6.1 Transactions . . . 47
6.2 Classification Constraints . . . 49
6.3 Object Constraints . . . 51
6.4 Cardinality Constraints . . . 52
6.5 Propagation . . . 54

7 The Persistent Store **59**
7.1 Features of the Persistent Store . . . 59
7.2 The Organization of the Store . . . 60
7.3 Streams . . . 61
7.4 The Disk Streams . . . 64
7.5 The Log-Structure . . . 67
7.5.1 The Header . . . 68
7.5.2 The OID Table . . . 69
7.5.3 The Objects . . . 69
7.6 The Persistent Store Manager . . . 72
7.6.1 Objects Allocation . . . 72
7.6.2 Objects Deallocation . . . 73
7.6.3 Waking up Objects . . . 74
7.6.4 Putting Objects to Sleep . . . 74
7.7 Caching Mechanism . . . 76
7.8 Packer . . . 76
7.9 The Root Table . . . 80

8 Transactions **83**
8.1 The computation of the validity envelope . . . 83
8.2 Validation . . . 84

9 Objects and Collections **87**
9.1 Collections . . . 87
9.2 Objects . . . 90
9.3 Operations and Transactions . . . 92
9.3.1 Loading and Storing DB-objects . . . 94
9.3.2 Loading and Storing collections . . . 97
9.3.3 Adding an object to a collection . . . 97
9.3.4 Removing an object from a collection . . . 98
9.4 Constraints and Update Propagation . . . 100
9.5 Object Evolution . . . 102
9.6 The Query Representation . . . 103

9.7 The Query Evaluation 105

10 Measurements and Performance Tests 109
10.1 Test Beds 109
10.2 Size of the System 110
10.3 Speed of the System 111

11 Summary, Conclusion and Future Works 113

A User Manual and Tutorial 117
A.1 Creating the Database 117
A.2 Inserting Objects 119
A.3 Searching Objects 120
A.4 Evolution 122

Abstract

Information, whatever its nature or its form, is an essential element of our society. However, this information is only useful if we are able to communicate it to other people and, above all, if we can store it in order to benefit the future generations.

During these past years, the Internet has contributed to the evolution of our means of communication in a considerable way; on the other hand, little progress has been made with regard to the long-term storage of this information.

This dissertation proposes a different approach to the problem of storing information thanks to a new database management system: *Odéon*. This system is built on a modern data model, which can represent not only information, but also the *evolution* of this information over the course of time.

Indeed, we know that the data we want to manage often has a very great lifespan. We also know that this data is not *static*, but that it evolves over time. The identity of an object will not change, but its *role*, its *classification* and its *attributes* will change.

Evolution is a significant aspect of information. However, few relational or object oriented models are able to deal with it in a simple and efficient way. The Odéon system proposes an effective solution to this problem.

In addition to evolution, we pay great attention to the speed and the efficiency of Odéon. The resulting system has excellent performance due to the Oberon language (in which Odéon has been entirely implemented), the effective use of the system's resources, and the persistent store adapted to our model.

Résumé

L'information, quelle que soit sa nature ou sa forme, est un élément essentiel de notre société moderne. Mais cette information n'est vraiment utile que si on est capable de la communiquer à d'autres personnes et, surtout, si on peut l'archiver, afin d'en faire profiter les générations futures.

Ces dernières années, Internet a fait évoluer nos moyens de communication de manière considérable ; en revanche, peu de progrès ont été faits en ce qui concerne le stockage de cette information.

Cette dissertation propose une approche différente au problème de l'archivage de l'information grâce à un nouveau système de gestion de bases de données : *Odéon*. Ce système repose sur un modèle moderne permettant de modéliser non seulement l'information, mais aussi *l'évolution* de cette information au cours du temps.

En effet, nous savons tous que les données que nous voulons gérer ont souvent une très grande durée de vie. Nous savons aussi que ces données ne sont pas *statiques*, mais qu'elles évoluent au cours du temps. L'identité d'un objet ne va pas changer, mais son *rôle*, sa *classification* et ses *attributs*, eux, vont changer.

L'évolution est une composante importante de l'information. Cependant, peu de modèles sont capables de la gérer simplement et efficacement, qu'ils soient relationnels ou orientés objet. Le système Odéon propose une solution élégante à ce problème.

En plus de l'évolution, nous avons mis beaucoup d'importance sur la rapidité et l'efficacité de notre système. Grâce au langage Oberon (dans lequel Odéon a été entièrement réalisé), ainsi qu'à une utilisation efficace des ressources et surtout à un système de stockage persistant adapté à notre modèle, nous avons réussi à atteindre d'excellentes performances.

Part I

Ideas and Concepts

Chapter 1

Introduction

I never waste memory on things that can easily be stored and retrieved from elsewhere *Albert Einstein*

Database functionality is necessary for a large category of modern computer applications. This is the case for not only traditional commercial applications such as payroll systems or personnel record management systems, but also for modern document management systems or computer aided engineering systems.

1.1 The Current Situation

Such applications are difficult to implement because they require a powerful *data model* for storing a large quantity of information and a powerful *operational model* to compute the complex operations required by the application. The traditional way to develop such *complex* and *data intensive* applications is to use a general purpose and powerful computer language together with an efficient database management system (DBMS). A typical example of such a combination is the C++ programming language and the Oracle relational DBMS. However, such combinations are seldom harmonious because of the following problems:

- impedance mismatch between the type systems
- impedance mismatch between the data models
- problem of object evolution

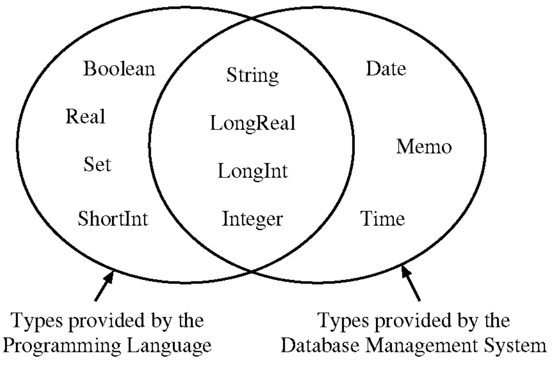
Boolean
Real
Set
ShortInt
String
LongReal
LongInt
Integer
Date
Memo
Time
Types provided by the Programming Language
Types provided by the Database Management System

Figure 1.1: impedance mismatch between the type systems

The first problem is that the type systems provided by the programming language do not necessarily match with the one provided by the DBMS. Some data types are only known by the programming language and some others are only known by the DBMS (Figure 1.1). The consequence is that the application must either use a reduced type system, or use a mapping mechanism between the different types.

The second problem is even worse and is about the mismatch between the data models. During the last years, programming languages have evolved towards modern programming paradigms like *object-oriented* programming [Mey88] or *component-oriented* programming [Szy98]. These new paradigms provide better encapsulation mechanisms and are the key for extensible and reusable software [GVJH94], [Fow96], [BMR95]. Database applications could also take advantage of these new paradigms, provided that the used DBMS also supports these paradigms, which is not directly the case with most relational DBMSs. However, some object oriented database management systems (OODBMS) like O2 or Objectivity already exist, but these systems are usually much less efficient than the relational DBMS. Another problem is that people are still used to handling their database problems in a “relational” way and it is hard to make them change their habits. And it is even harder if the new technique is less efficient than the one they are using!

The last problem is the problem of object evolution. In programming languages, the type system is usually a *static* structure, that is that an application is normally not able to dynamically create new types or to change existing type definitions *at run time*. This is not a problem for most applications, but it turns out to be a severe problem with database applications. Actually, the information stored in a database usually has a longer life time compared to that of programs and this information usually needs to evolve with time. Object evolution occurs, for example, when a student becomes an assistant or when a person starts playing tennis. In the first case, the object loses the attributes relative to students and gains those relative to assistants. In the second case, the object only gains the attributes relative to tennis players. Expressed with the terms of programming languages, we can say that objects need to be able to change their *type*, to gain new ones and also to lose some.

If a DBMS wants to represent the reality, and this is usually the case, then it needs to support object evolution and the programming language that manages the information has to be able to deal with these evolving objects.

1.2 Why Odéon?

The goals of the research project presented in this dissertation are the following ones:

- Implement an OODBMS that is harmoniously integrated in a programming language
- Implement an *efficient* OODBMS that can compete with RDBMS in terms of speed and memory consumption.
- Develop a data model in which object evolution is possible and even straightforward

In order to achieve these goals, we decided to implement our system in Oberon. Actually, Oberon has many features that makes it ideal for such a system; it is small and efficient, object-oriented, extensible and it can be used for system programming as well.

Our system has no impedance mismatch between the type systems because our DBMS is written *in* Oberon *for* Oberon. Odéon is object-oriented, so there is no impedance mismatch between the data models either.

We will also show in the dissertation that our system is efficient in terms of speed and memory consumption. We have thus achieved a *harmonious* and *efficient* integration of a OODBMS into the Oberon system.

1.3 Contributions

Thanks to the *log-structured* persistent store that we have implemented, Odéon is efficient, it uses only few resources and is well integrated in the hosting operating system. With this system, we have demonstrated that an OODBMS can be as fast and as efficient as a RDBMS. The problem of some OODBMS is that their persistent store is built on top of an RDBMS like Oracle or any other SQL based database system. We think that is not the right way to go because all these layers use a lot of memory and the conversions between these layers is a major cause of inefficiency.

Besides this, the data model that we have chosen for our system shows an interesting solution to the problem of object evolution. Actually, the approach we took was to unify the concept of *types* and of *collections*. In many OODBMSs, each object is of one or more given types and is also member of one or more collections. In our data model, we do not have this dual classification; expressed in a simple way, we can say that in our model, the collections also have the role of defining the type of its members. We admit that the data model itself cannot be seen as our contribution because it has been mostly inspired by the existing OM model. Our contribution is rather in the change that we did to the relation between the types and the collections defined by the model and the consequence of this change in relation to the problem of object evolution.

The first two contributions are not specific to the Oberon system. In fact, our persistent store could be ported to other platforms and the data model is in no way dependent of the underlying system. But we will see in this dissertation that Oberon is an excellent language and an excellent system for supporting such an object oriented database management system. Odéon is a good example of the flexibility and the extensibility of the Oberon system, especially in the field of database systems. Further, Odéon can be the basis for many interesting research areas in DBMSs.

1.4 Structure of the Thesis

The first part of this dissertation explains in detail the data model supported by our system, with emphasis on its powerful classification and evolution mechanisms. Evolution is very important because, as we cannot predict the future, we cannot know in advance all the roles that the object may play during its life time. And even if we could predict the future, we still would like to have an evolution mechanism because we may not want to always store the whole history of all objects, but just the current status.

The second part of this dissertation shows the implementation of our DBMS called *Odéon*. This implementation has been made in the HP-Oberon system [Sup94], using the Oberon-2 language [RW92]. We tried neither to change the Oberon language, nor the Oberon system, but only to *extend* the latter with a DBMS.

Chapter 2

Background

2.1 The Persistent Store

We have seen in the introduction that our system is an object-oriented database management system. In other words, our system has to store and manage some pieces of information represented as objects. The storage of objects is therefore a fundamental aspect of our system.

When we want to store information for a long time, and this is the case with our system, we first need a so called *persistent store*. A persistent store is a storage system where the information will still be present if the system is powered off. The persistent store also provides all the facilities needed for adding new objects, removing, updating, and searching objects. In order to achieve this goal, we have many solutions:

- Use a standard relational database management system
- Use an existing object oriented persistent store
- Use of a persistent programming language
- Develop our own persistent store as a set of modules

The first option, that is the use of a standard database management system, is probably the simplest one. With this approach, we have to transform our objects into a form that can be managed by the underlying DBMS, and this is precisely the weakness of this model. It is as using small envelopes for storing big sheets of paper; you will always have to fold the paper before putting it in the envelope and unfold if after having taken it out. This transformation is very costly and is a principal source of inefficiency in the system using it. However, this solution could be used for prototyping, where it is more important to quickly have a working system than to have a fast system.

The second approach also consists of using an existing system for storing the data, but here, the storage is already an object oriented store like *Objectivity* or *ObjectStore*. In this case the transformation of the object is much easier or may even be unnecessary. These systems usually support the object oriented languages C++ and Java, they usually also have a standard SQL interface and some of them also support other languages like Smalltalk. But none of them support the language Oberon, so if we want to use one of these systems with Oberon, we still will have to transform our objects and we will lose efficiency. Another problem with these systems is that they are quite complex and they require a lot of memory and disk space. This is against the philosophy of Oberon: what we want is a small, simple, and efficient system. However, for commercial applications, having to deal with huge amounts of data and running on powerful computers, these storage systems are quite attractive.

The third approach is to use a persistent programming language. But we want to do it with Oberon, which is not a persistent programming language, so the solution would be to extend the Oberon language with a persistent store. This has already been done with the Algol and Pascal languages. Algol has been extended to PS-Algol [ABC$^+$83] (which led to Napier88 [MBCD89]), a programming language with everything needed to manage persistent objects. In PS-Algol, the persistence is *orthogonal* to the type system of the language, that is every object can be either transient or persistent and we can use them in exactly the same way. PS-Algol has not been designed to provide direct support for a specific data model but provide programmers with the tools they need to define their own one. The Pascal language has been extended to Pascal-R [SM80]. In the name, the "R" stands for *relational*, so the goal of Pascal-R was not to provide orthogonal persistence, but to provide special types to allow the construction of persistent tables and to provide the functions for managing these tables. So in Pascal-R, the language has been extended to provide direct support for the relational data model. This approach is interesting because it completely removes the impedance mismatch between the programming language and the persistent store. This means that the objects can be stored and restored without any transformation. These resulting systems are usually small and efficient. This sounds very attractive, but in our point of view, the management of persistent objects is not the role of a programming language. Actually, in our point of view, a programming language has to be as simple as possible, and if we need special data types or extra functions, this has to be done using libraries or, in the Oberon terminology, using modules.

The last option was to develop our own persistent store using a set of modules. This option is possible with Oberon because the system is open and easily extensible. In Oberon, every subsystem is already implemented using modules; the file system, the network system, the printing system, etc, are all implemented using modules. It is then natural to implement the persistent store in Oberon using modules as well and to leave the Oberon language unchanged. This option has also been chosen in other projects of extending Oberon with persistent store: The Oberon System-3 [FM98] for example uses a persistent store as an alternative to files for storing objects. It this system, it is possible to define a *library* [FM98, Page 231] that can be viewed as an object container. Each object that is bound to a library can be made persistent by storing the whole library content on a disk. The problem with the Oberon System-3 is that it cannot read individual objects from the library; it has to read all of them. This system is simple and efficient for many applications but is not applicable to databases where the amount of data is usually much bigger that the main memory available in the system. Oberon-D [Kna96] is another extension of Oberon with persistent store. In this system, the persistence is orthogonal to the type system and is totally transparent for the application. The system offers good performance, is easily portable and is quite simple. We could have used it for our project, but Oberon-D has no support for versioning and has no transaction/roll-back mechanism. We could have extended this system for our purpose, but it is not sure that we would have been able to achieve our desired performance. For these reasons, we decided to implement our own persistent store, specially tailored and optimized for our needs and for our needs only.

2.2 The Object Oriented Data Model

A *data model* is a set of constructs and operations used to specify how information is represented in a database. The model underlying our system will be presented in detail in the following chapters of this dissertation, but in this chapter, we want to show how our model places itself among the other known models, what it has in common with them and also how it differs from them. In other words, we want to show the “spirit” of our model and of the resulting implementation in the Odéon system.

The model presented in this dissertation has been mostly inspired by the *OM* data model conceived by M. C. Norrie [Nor93]. *OM* is a simple, generic, object oriented data model in which the concept of *classification* of objects is of primary importance. The basic classification structure in *OM* is expressed in terms of *collections*. Our system also uses collections for the purpose of classification, but the main difference between *OM* and our model is in the concept of type and its relation between objects and collections.

Most current object models do not distinguish clearly between typing and classification. Some have only a single constructor – that of *type* – which primarily serves the purpose of describing objects. The grouping of objects is a consequence of the typing because types also automatically maintain an extent with all objects of that type. So these systems focus on types and rely on bulk constructors such as set, bag, etc., to form semantic groupings, but have no notion of sub-collections, classification constraints, etc.

In *OM* there is a clear separation between type and classification. Types are used to define the attributes of the objects and collections are used to build classifications. There is no direct relation between the two concepts. Because the classification is no longer a consequence of the type, it can be much more powerful.

Contrary to most current object models, in Odéon the *type* of an object is a direct consequence of its *classification*. If we group some objects in a collection, then these objects have some common attributes, and these common attributes define the type of the object. An object can be in many collections, and in each collection it is of a corresponding type. If an object is in no collection, it is an abstract object and has no attributes. It represents only the *existence* of that object.

Let us illustrate this principle by an example: consider a textbook. If a student wants to describe his favorite textbook, he will probably give the title of the book, the author, and maybe the publisher and the year of publication. He could also give a small abstract and his personal comments about the book. On the other hand, for the person in charge of the archive of a library, the same book will be described by its ISBN and maybe also its size, so that he can store it and find it in a more efficient way. For a typesetter, the same book may be described by the color of the cover or by the fonts used in the book.

As we have seen in this example, the *type*, or the properties of a book cannot be defined without giving the context in which the book is considered. In our model, the collections are used to group objects and also to define a context, and the simple fact of inserting an object in a collection automatically defines the attributes corresponding to the context represented by the collection. The object must not *be* of a given type to be in a collection, it *receives* its type when it is inserted in the collection.

In *OM*, the classification in collections is a very dynamic structure, that is, objects are often inserted in or removed from collections. However, the type of an object is more static and special operations (*dress* and *strip*) are required to change it. In most programming languages, the situation is the same: we can build dynamic structures like lists or trees of objects, but the type of the object is usually immutable. However, in reality, the type of an object is seldom constant and objects are usually in many different contexts during their lifetime. A person, for example, first begins by being a child, then goes to school and is thus in the context of a student. Later the same person may become an assistant, then maybe a professor, and so on and so forth. Most current models have serious problems to represent such a dynamic life. *OM* can do it and our model gives another solution that is simple and efficient to this problem.

As we have said, the model used in our system has been mostly inspired by the *OM* data model. It is then natural that our system has some similarities with other systems based on *OM*. We can compare our system with the two other implementations of *OM* made at the ETHZ. The first system we want to compare with is *OMS Pro* [NW99], [KNW98], [NW98]. This system has been written in prolog and uses the persistent store provided by *Sictus Prolog*. OMS Pro is an accurate implementation of the *OM* data model and it also supports object evolution. In OMS Pro, an object can also have many types and it can be in many classifications. OMS Pro also has a query language, AQL, that allows complex queries to be expressed and to fully exploit the power of the algebra behind the *OM* model.

The main drawback of OMS Pro is its lack of efficiency for handling large collections. But this has never been the goal of OMS Pro, actually, OMS Pro has to be seen as a *prototyping system*, with which we can experiment and play with the *OM* model. OMS Pro has also an integrated graphical user interface, which even facilitates its use. OMS Pro is a perfect system for education and for small databases. It allows an object oriented database to be rapidly set up and managed through the AQL language.

Another system that implements the *OM* model is the *OMS Java* system [SKN98]. As his name indicates, this system has been written in Java and is intended for the Java community. OMS Java is both a general data management system and a framework for the development of advanced database application systems. To illustrate the extensibility of the system, OMS Java has been successfully extended to a *temporal* object-oriented database system.

OMS Java is a very interesting implementation of the *OM* data model, but, as it is the case with OMS Pro, the system is not very efficient for handling large collections. The problem here is with the persistent store. Actually, OMS Java, in its current version, can choose between two persistent stores: the first is using the JDBC interface and any relational database system (like ORACLE) and the second is using a commercial product from Object-Store. The first version is not very efficient due to the impedance mismatch between JDBC and the *OM* data model. The second version is much faster but not so flexible. It requires for example that the data is stored locally on the machine which runs OMS Java. The problem is not trivial and some work is still done to improve the efficiency of the persistent storage of OMS Java.

Outside the *OM* community, other systems propose a solution to the problem of object evolution. For example, in 1993, Ghelli, Albano, Bergamini and Orsini, presented an object oriented database programming language called “Fibonacci” [ABGO93]. In this system, as in our system, objects have the ability to dynamically change their type while retaining their identity. This is implemented using the concept of *roles*. In Fibonacci, an object can play many roles, and in each role, it displays different attributes and different methods. The Fibonacci language has simple constructs to manage these roles; it is easy to make an object playing new roles, to change the roles of an object or to remove roles from an object. All these operations are done dynamically, that is, while the system is running.

In Fibonacci, there is no impedance mismatch between the language and to persistent store because Fibonacci is a database programming language, this means that the database functionality, and thus the persistent store and the evolution mechanism are integrated in the language itself.

In terms of evolution and roles management, Fibonacci has the same features as Odéon. The advantage of Fibonacci is that the integration of the database functionalities and the programming language is even better because the database functionalities are part of the language. The advantage of Odéon towards Fibonacci is its legacy from OM regarding its powerful collections and constraints model. The other advantage of Odéon is the fact that it is not a new language, but it is an extension of Oberon. By this fact, it profits from the power of the Oberon language and from all libraries and extensions available for Oberon.

Another interesting approach is the one presented by Gottlob, Schrefl and Röck [GSR96]. They were also unsatisfied by the fact that in many object oriented systems, the association between an object and its type was *exclusive* and *permanent*. They wrote that : "...these systems have serious difficulties in representing objects taking roles over time." In their article, Gottlob, Schrefl and Röck describe a model where the *type* hierarchy is extended by another hierarchy, namely the *role* hierarchy. Their approach is interesting because they did neither invent a new language, nor modify an existing one. On the contrary, they built a new structure on top of an existing system. From this point of view, their approach is very similar to ours. The difference is that they did not *replace* the type hierarchy but they *extended* it with roles. In their model, an object has one static *type* and one or more dynamic *roles* (objects can acquire and abandon roles dynamically). In our system, we reconsider a part of the object oriented paradigm and we used the classification to define the type of the objects.

Another feature of the system of Gottlob, Schrefl and Röck is that objects may occur repeatedly in the same role. For example, an employee may become a project manager of several projects. In Odéon, we do not have this feature. However, this sort of problems can be solved with the concept of *associations* defined in the *OM* model.

In order to demonstrate the practicability of their approach, they have implemented their model in Smalltalk. But their ideas are not limited to this language and can be applied to any language based on Smalltalk.

To finish this chapter, we also would like to compare our system with O_2. O_2 is an OODBMS that comes in different flavours, depending on the underlying programming language. The first two variants have been CO_2, based on the C programming language, and $BasicO_2$ based on the Basic programming language. In the first version of O_2, there was no support for object evolution, but a newer release of the system now fills this gap. A new system method called *migrate()* allows an object to migrate from its class to any of its subclasses. This is better than nothing, but we consider too restrictive the fact that the migration is only allowed to the subclasses. In O_2, contrary to the *OM* model and the different implementations of it, we are also missing the rich classification scheme and the constraints ensurance mechanism.

However, O_2 has a good graphical user interface and allows to rapidly set up an object-oriented database system and then to develop the application that will use this database.

Chapter 3

The Data Model

The role of a database system is to store and manipulate information about the application domain. To specify how this information is mapped in a database, we need to specify the constructs and the operations of the database system. These constructs and operations are specified in terms of a *data model* and this chapter describes the one underlying our system.

The model we choose has been largely inspired by the OM model [Nor92] of M. C. Norrie. OM can be briefly described as a generic, object-oriented data model, offering a rich classification scheme, object and schema evolution, and strong constraint ensurance. In this context, the term "generic" means that the model can be used for both conceptual modeling of the application domain and for the implementation of the resulting database system. This is a considerable advantage over the traditional method, where the design is usually done with the Entity-Relationship model and the implementation is then done with another model, namely the relational model.

In the following sections, we describe in which ways our model is *object-oriented* and what the concept of *classification* means. We also explain the concept of *Typing by Classification* which makes our model distinctive from OM and also from most currently existing models. The evolution and constraint ensurance mechanisms will be explained in the following chapters.

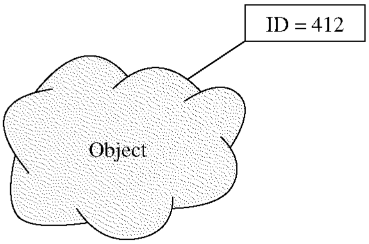

Figure 3.1: The Object is associated with a unique Object Identifier (OID)

3.1 What is an Object?

We said that our model is "object-oriented", but as this term is still used in several ways, we start this chapter by stating what "object-oriented" means in our model. Therefore, we first need to explain what an *Object* is.

Basically, an object, in our model, is just a representation of a thing (concrete or abstract) or of a being, inside a database. In other words, we can say that:

> The primary role of an object, is to *represent*
> or to *map* something from the world
> of the application domain into a database.

It is obvious that this representation has to be unequivocal in the database, because we must always be able to *identify* the objects and to differentiate each of them from the others. At this point, we cannot use the attributes of an object to differentiate it from the others because, as we explained in the preceding chapter, the objects only have attributes when they are in a given context, and also because we do not want to prevent two different objects from having exactly the same attributes in the same context. It is quite possible to think of a library for example, where all copies of a book are represented by the corresponding number of objects; in this case, all objects will have the same attributes, but they still are different objects. The obvious solution for solving object identity is to assign a unique object identifier (Figure 3.1), or *OID*, to each object. This OID is assigned by the system and not by the user of the system, so we can guarantee that OIDs will be unique.

We do not need to specify the type OIDs, and actually, the only requirement for OIDs is that the system must be able to determine if two OIDs are *equal* or not

Having defined this last point, it is now possible to know when two objects are equal, namely:

> Two objects x and y are equal, if
> and only if their OIDs are equal.

Another important characteristic of OIDs is that they are guaranteed to be an absolutely immutable attribute of the object: when an object is created, it obtains its OID, and it keeps it during its whole life. Neither the user, nor the system must be able to change the OID of an object. Only when the object is wiped out of the system, then, and only then, its OID may be recycled and reassigned to a newly created object.

An abstract object that carries just an OID as information, is not very useful for an application, but we will see later in this chapter how it is possible to define a context for the objects, and how to set attributes to the objects that exist in such a context.

Object identity is certainly an important feature of an object-oriented database management system, but it is not the only one: contrary to the relational DBMS, OODMBS must also be able to manage objects that are more complex than simple flat records. Such complex objects arise not only in large CAE (computer aided engineering) databases, geographic information systems or multimedia databases, but also in modern payroll systems, library management, and many others. In fact, until now, we were not used to use complex objects for the modeling of such application models because we did not have the systems for supporting such complex objects. Now that object oriented systems are available, we can rethink the problem and try new models that will be, we hope, cleaner, simpler and better than the previous ones.

Another feature of OODBMs is that objects should not remain passive. On the contrary, they must be *active* in the sense that they should be able to accept *messages* and respond to those messages by calling a routine or, in the object-oriented terminology, a *method*. In our system, this concept of active objects is provided in the same way as in Oberon, namely by using the message handler metaphor [RW92, Section 12.5]. We chose this solution because it is very simple, and powerful enough to show the principle of active objects. But nothing prevents the system from being extended with a more complex method calling mechanism.

A last important feature of an Object Oriented system is the support for *inheritance*. With inheritance, we mean a mechanism that allows a class (or,

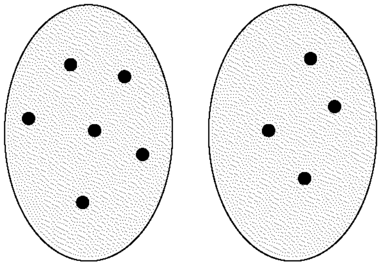

Figure 3.2: A Simple Classification

in Oberon terminology, a type) to be defined as an *extension* of a previously defined one [RW92, Chapter 11]. Section 3.4 will show how inheritance is managed in our model, namely through *ISA* relations.

3.2 Classification by Collections

The primary role of what we call *classification* is to organize and to structure a database by *disposing* or *classifying* objects in different *collections*. In our model, a *collection* represents a *semantic grouping* of objects, i.e. a set of objects with a common role and common properties. A collection is an object itself.

For example, a collection *Professors* could contain a set of persons giving a course in a university, and another collection *Students* could contain a set of persons attending these courses. Figure 3.2 shows a graphical representation of such a classification. In this figure, collections are represented by grey ovals and the objects inside collections are represented by black dots.

In our terminology, the objects that belong to a collection are called the *members* of the collection. The number of members of a collection C is called the *cardinality* of C and is denoted by card(C).

The two basic operations that are applicable to collections are *insertion* and *deletion*. The semantics of these functions is obvious: insertion adds a new object to a collection and deletion removes one from it.

From a formal point of view, the insertion operation can be seen as a function with the following property:

$$C' = ins(C, o) \quad \Rightarrow \quad C' = C \cup \{o\}$$

where $\cup$ is the union operation [SM77, Page 85]

And the deletion operation:

$$C' = del(C, o) \quad \Rightarrow \quad C' = C - \{o\}$$

where $-$ is the difference operation [SM77, Page 85]

From the description we gave and from the representation we made in figure 3.2 it seems that collections are identical to *sets* in discrete mathematics, but we will see in the following chapter that collections are not only used to group objects (as sets do), but that they are also used to define *attributes*, or in other terms, the *type* of their members.

However, if we only consider the role of grouping objects, collections are, in fact, the same as sets. In a standard textbook about discrete mathematics [SM77, Page 75] we can read the following definition:

> "A *set* is any collection of objects which can be treated as an entity, and an object in the collection is said to be an *element*, or *member*, of the set."

If our collections can be treated as *entities*, then this definition also applies to them. And actually, in our model, collections are themselves objects, thus they can also be treated as *entities*.

Since collections are objects, it is even possible to make collections of collections and allow nested structures of classification to be built.

The set of all collections in a database, together with the rules that control these collections, is called the *schema* of the database. In many available DBMSs, a schema is a rather rigid structure in the sense that once it is defined, the schema is never changed. Here, we have a much more flexible model. Actually our model allows collections in the database to be added or removed, or in other terms to change the schema of the database, while the database is in use. The ease with which one can add new classifications leads to new ways of designing database schemas. Let us take an example to illustrate this idea. Suppose that we want to design a biochemistry database for storing information about proteins and that we are mainly interested in the classification of the species to which these proteins belong. The usual way to model this is to define a collection of proteins, a collection of species and a relation* between these two collections to show which protein belongs to which species. Figure 3.3 shows such a schema.

*See Section 3.5

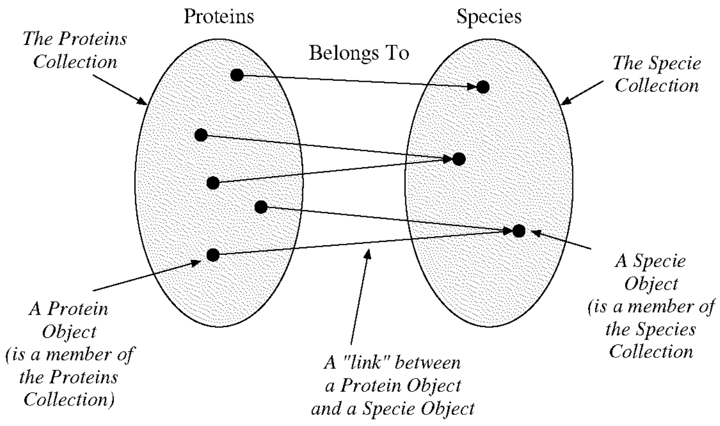
Proteins
Species
The Proteins Collection
Belongs To
The Specie Collection
A Specie Object (is a member of the Species Collection
A Protein Object (is a member of the Proteins Collection)
A "link" between a Protein Object and a Specie Object

Figure 3.3: Simple Biochemistry Database with Collections and Relations

From another point of view, species could also be considered as *collections* containing proteins, as shown in Figure 3.4. Actually, we say that a protein *belongs* to a species and the relationship between proteins and the species is the same as the one between objects and collections. In fact, this second schema is equivalent to the previous one in the sense of the information it contains. The only difference is that we replaced the species objects of the first model by collections and we changed the explicit "belongs to" relationship between proteins and species by a "member – collection" construct.

However, this second model is only possible if the underlying system enables dynamic schema extension. We do not claim that, in this example, the second design is better than the first one; we only show that our model provides this new possibility to give the user a broader choice of approach to develop the best suited model for his needs.

In the introduction of this chapter, we said that our model supports a *rich classification scheme*. This means that a given object can belong to more than one classification (or collection) at the the same time. For example, in the university model that we outlined just before, we may also be interested in sport activities of the persons. An object could be at the same time in the *Professors* collection and, for example, in a *Tennis Players* collection. In fact there should be no limit in the number of collections in which a given object can be. As we will see later in this chapter, our model fully satisfies

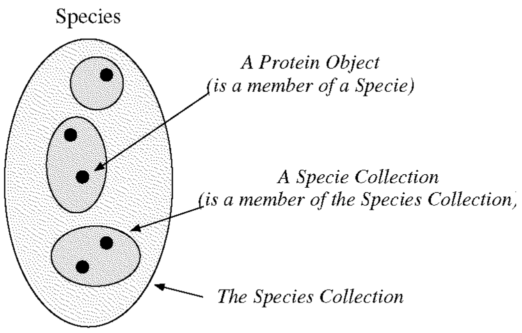
Species
A Protein Object
(is a member of a Specie)
A Specie Collection
(is a member of the Species Collection)
The Species Collection

Figure 3.4: Simple Biochemistry Database with Nested Collections

this requirement.

Several OODBMSs do not restrict their collections to be *Set-Collections*. They also allow *Bag-Collections*, in which an object can have many instances, or *List-Collections* in which objects are ordered. This may be useful for some specific applications, but this also adds considerable complexity to the system. On top of that, the algebra for bags or for lists is not as well established as the one for sets. For example in context of lists, what is the intersection between the list $[4, 5, 3, 1]$ and $[3, 4, 7]$? Is it $[3, 4]$, $[4, 3]$, or even something else? So whereas we do not have a consistent, unambiguous and deterministic algebra over all sorts of collections, and because we want to keep our system small and simple, we decided to provide the well known set-collection only in our model.

3.3 Typing by Classification

Before starting on this central point of our model, let us answer the question: "Why do we need Types?". The usual answers to this question are

- To check the compatibility of objects
- To group objects that have the same role in the application together
- To associate attributes with objects

Checking the compatibility of objects is principally a feature of the *operational* (or execution) model and not of the *data* model. Actually, an object must be compatible with another only when it participates in an *operation* with the other object. This is for sure an important aspect, but for the moment, let us concentrate on the data model.

The second point is the use of types for building groups of objects. This point is still more relevant for a data model. We have seen in the previous section that in our model, grouping is already done by collections, so this point is already available in our model.

The third and last point is the most interesting one for us. Actually, our model does not yet allow to associate attributes with its objects and this makes our model not so attractive yet. Instead of building an additional type system, we decided to use the existing collections model and to expand the role of collections in such a way that they will also be used to associate attributes with their members.

This is, in fact, a logical consequence of the notion of collection as a semantic grouping. Actually, a collection gives a *context* or defines a *role* for the objects it contains, and within such a role, all the objects share some common *attributes* or *properties* that are related to that role. For example, all members of a *Professors* collection could have an attribute such as the name of their university or the number of courses they are giving. Our model uses this issue to effectively associate attributes to objects. We say that a collection *defines* attributes, and such attribute definitions are characterized by a set of tuples $\langle \mathtt{name}, \mathtt{type} \rangle$.

The set of all attribute definitions given by a collection is called the *characteristic* of the collection. When an object o is inserted in a collection C, it becomes a member of C, and by being a member of C, it also receives all the attributes that are defined in the characteristic of C.

This idea of using collections to assign attributes can be summarized as *Typing by Classification*, which is contrary to the well known idea of *Classification by Typing*. Indeed, in most object-oriented models, typing is used to represent a classification. The novelty in our model is that it takes the opposite approach, i.e. using the classification, or in concrete terms the collections, to define the attributes of the objects. Another motivation for such an approach is that it allows to *dynamically* change the classification and change the attributes of the objects, without having to recompile the application. *Types* are therefore *dynamic*. We will study this feature called *evolution* in more details in the next chapter.

This construction of roles using semantic grouping is an interesting alternative to the other solutions found in the literature such as in [ABGO93] or in [GSR96]. In the first example, the concept of evolution has been implemented in "Fibonacci", a new language especially designed for solving evolution. In the second example, the authors show an implementation of a role hierarchy for evolving objects. This implementation is done with Smalltalk.

From this point of view, collections are somehow similar to Oberon *types*. The main difference is that, in Oberon, an allocated object has only one single type and it keeps this single type during its complete lifetime. There is no way for it to dynamically gain new attributes. In our model, an object can be in many collections at the same type and it can also dynamically move from one collection to another and in each collection, the object assumes the attributes relative to the role represented by that collection. The challenge was then to integrate this highly dynamic type system into the Oberon system, but this point will be discussed in the second part of this dissertation.

When we described the attributes, we did not specify the form of their *types*. In fact, this is not relevant for the formal definition of the model given here. We will see later that, as our implementation has been programmed in Oberon, and as our system is embedded in the Oberon system, we also naturally choose the Oberon types for the formal attributes of Odéon. This makes our model much more flexible than the relational model in which the first normal form [EN94] disallows multi-valued and composite attributes.

3.4 Inheritance and the ISA Relation

Within the Oberon language, inheritance is carried out by using the principle of *Type extension*. In concrete terms, this means that in Oberon, a record type can be declared as an *extension* of another record type and therefore, in addition to the fields declared in this new record, it also *inherits* all the field declarations of the record it extends. Figure 3.5 shows an example of a *professor* record, defined as an extension of a *person* record.

Inheritance is a widespread construct in computer languages, but it is also present in modern database modeling. For example the "EER" (Enhanced Entity-Relationship) model [EN94, Section 21.1] includes the concepts of sub-class, super-class and attribute inheritance. Our model also provides inheritance by the way of the *ISA* relationship. The *ISA* relationship is, in the OOP terminology, a *class/sub-class* relationship and is equivalent to the relationship between an extended record and the record being extended (called *base record*) in Oberon. As we have seen in the previous sections, in our model, classes are represented by collections, so we can say that

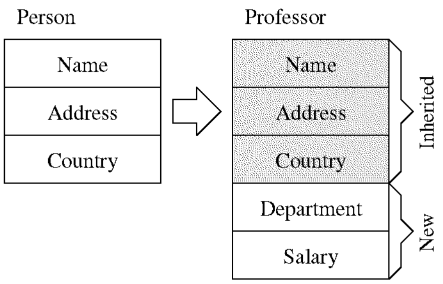

Figure 3.5: Type Extension in Oberon

our ISA relationship is rather a *collection/sub-collection* relationship. To compare with Oberon a *super-collection* in our model is equivalent to the base-type of an extension and a *sub-collection* is equivalent to the extension itself.

In the graphical representations of schemas, the ISA relations are symbolized by a wide arrow. Figure 3.6 shows such an ISA relation.

A *sub-collection* S of C can be defined as a collection whose members are a subset of those of its *super-collection* C. A sub-collection S also inherits all attributes defined in its super-collection C. The sub- super-collection relationship between S and C is noted $S \leq_c C$.

The sub-classing relation $\leq_c$ on collection defines a partial ordering of the collections of a schema. In this partial ordering we define the *direct* sub-classing relation as follows: S is a *direct* superclass of X if S is a superclass of X and there does not exist any other superclass T of X that is also a subclass of S. We write $S = \uparrow X$. In a formal way, we can write

$$S = \uparrow X \Leftrightarrow X \leq_c S \text{ and } \nexists T \mid X \leq_c T \text{ and } T \leq_c S.$$

If a collection S has no superclass, then $\uparrow X$ is defined as a pseudo collection called *universe* and noted $\mathbb{U}$ which provides no attribute and that contains all the objects of the database.

Like the Oberon language, our model does not support *multiple inheritance*. This means that a collection can have at most one super-collection. Single inheritance has many advantages over multiple inheritance, such as the absence of name clashes, no “lattice” structure, and less run-time penalty. See [Mös95, Section 8.6] for further details about these problems and about their solutions.

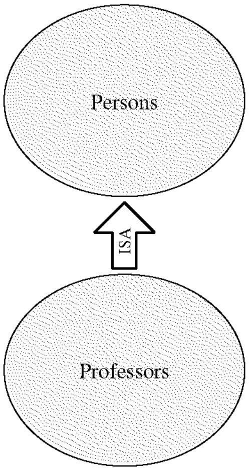

Figure 3.6: ISA Relationship

3.5 Associations

In the previous sections, we discussed collections and their elements, but these collections have been considered as independent entities. In many application domains however, we are not only interested in the classification itself, but also in the *relations* that exist between the collections. For example, in our university model, if we have a collection of *lectures* and a collection of *students*, we will certainly also be interested in information such as "which student attends which lectures", or in other words, in a *relation* between the collections *students* and *lectures*. In our model such relations are represented by what we call *associations*. An association is always *bound* to two collections (which can also be the same) and all pairs of associations must link objects between these two collections.

Since associations are collections themselves, they have all the characteristics of the collections. This is interesting, because it is then also possible to define attributes for an association. This feature, which is not available in the original model [Nor92], can be used in many situations, like in the following example: Suppose that we want to model the database of a computer hardware manufacturer. In this database, we need to store the *parts* that are used to build machines. Some of these parts are *basic* and some others are *composite*, i.e. they are made of several other *sub-parts*, which may again

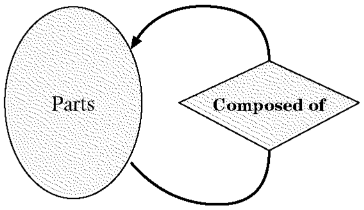

Figure 3.7: Association Between Parts and Sub-Parts

be either basic or composite. So the natural way to model this example is to have a collection of *parts* and a relation *composed of* that links parts with their sub-parts. Figure 3.7 shows such a schema. Note that in the graphical representation of the schema, associations are represented by diamonds.

This model works well unless we have parts made of many identical sub-parts. For example, if the main board of a computer is composed of four identical memory modules; this would result in having four distinct memory module objects in the collection and a separate link for each one. This is possible, but would not be very practical when a part is made of thousands of identical sub-parts. Fortunately, *attributed links* provide an elegant solution by having only one single memory modules object, a single link, and by assigning the number of same parts as an attribute of the link.

Another important characteristic of associations is that they have a direction: a pair of a given association A always goes from a *source* collection (noted $\overleftarrow{\mathrm{A}}$) to a *destination* collection (noted $\overrightarrow{\mathrm{A}}$). In other words, we can say that an association is a collection of ordered pairs.

In our notation, pairs are denoted $\langle \mathrm{s}, \mathrm{d} \rangle$, where s represents the object from the source collection and d represents the object from the destination collection.

Our model supports only *binary* associations, or in database terminology, our relations only have a *degree* of two. But since it is always possible to represent associations with more than two members using many binary associations, this restriction does not limit the expressiveness of our model. The standard literature, such as [EN94, Section 3.6] explains in details how to convert a relation of degree n using several binary relations.

People who are familiar with DBMS may be willing to read something about the "cardinality" of our relations. This subject will be developed in details in Chapter 6, where constraints are discussed.

3.6 The Root Objects

In this chapter we defined the different components of our model. We explained what objects are and how they are identified by an OID. This OID is for the moment *the only way* we have to find the objects in the database. Using OIDs is fine for the system, but they are not very well suited for a human user.

Actually, if the user needs a specific object, he want to access it by something more convenient than a system generated identifier! The problem of finding objects is somehow similar to the one of finding a file in the file system of a traditional operating system. In such file systems, the problem has been solved by using a *directory* where each file can be associated with a name identifying the file in question. Then, to access the file, we do not need to address its first sector (which would correspond to the OID in our case), but we specify the name we chose for the file and the conversion from the name to the first sector is then automatically done by the system.

In our model, the problem is very similar, and so is the solution we chose. We also provide a way to associate names to objects and we have a function that, given a name, returns the OID of the corresponding object. Our root table does not allows aliases; that is, it is not possible to have more than one name associated with each object. On the other hand, it is possible (and this is even usually the case) that an object has no associated name. In this case, the object is anonymous and can only be retrieved if it is member of a collection.

As we will see in the second part of this dissertation, we used a specially optimized construct to implement roots, but conceptually, this can be viewed as a special collection, called *Roots*, with a *predefined* fixed OID so that this collection can be directly accessed. This special collection defines only one attribute, namely the *name* of the object. Then in order to associate a name with an object, we just have to insert the object in the *Roots* collection and, through this collection, assign a value to the name attribute that is provided. Using the query algebra that we will describe in Chapter 5, it is then easy to look up a given root object by its name and thus, to obtain its ID. This root-table is similar to the one used in PS-Algol[ACC81].

Chapter 4

Evolution

A database system is seldom an immutable information source. The most frequent changes in a database system are done at the level of the collections: new objects are added to the collections and those that are no longer valid are deleted. These operations have been discussed in the preceding chapter and now, we want to show two other types of changes that also occur in a database system, namely the *object evolution* and the *schema evolution.* In our terminology, evolution means the ability that an object or a database has to adapt itself to a new situation. This ability is one of most important features of our model.

In this chapter, we give the conceptual details about evolution and its role in our model. To illustrate these concepts, we will give examples using a simplified database that could be used to manage people in a university. Figure 4.1 represents the schema of this database.

4.1 Object Evolution

The first form of evolution we want to show here is the *Object Evolution.* It represents the ability of objects to evolve over time. In this form of evolution, the schema of the database remains unchanged and no object is created or deleted. Actually, the changes that occur in object evolution affect only the roles that existing objects play in a database. The mutations that an object can undergo by the object evolution are the following ones:

- Gain of a role

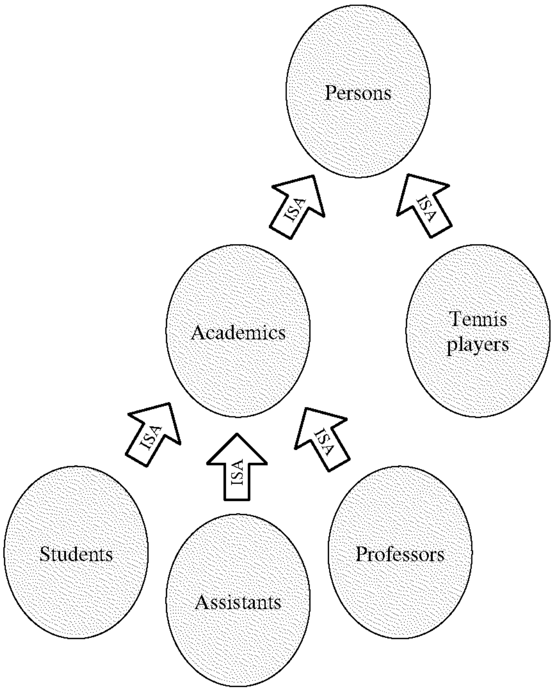

Figure 4.1: The University schema used in the examples

- Loss of a role
- Change of a role

The first mutation is the *gain* of a role and is also called *Dress* in OMS. In this case, an existing object has to play an *additional* role in the database. In our university model, such a situation could occur, for example, if a professor decides to start playing tennis. The object representing this person was already playing the role of a professor and now it also has to play the role of a tennis player.

The second mutation is the *loss* of a role, also called *Strip* in OMS. This form of evolution is the opposite of the previous one and occurs when an object *ceases* to play a given role. Let us give an example within the university model: after many years, our professor becomes too busy for playing tennis and decides to cease playing this sport, so he loses the role of tennis player.

The third and last mutation is the *change* of a role. This is, in fact, a combination of the two previous mutations. Here the object is simultaneously losing a role and gaining another one. For example, in our university schema, an assistant could be promoted to the role of a professor. In this case he loses his role of an assistant and gains the one of a professor.

In many systems, object evolution is a very complex operation because changing the role of an object also means changing its context, which in turn means changing its attributes and which in turn means changing its type. Because many systems use the type system of the hosting programming language, they are usually neither able to dynamically add new types to the database system nor to dynamically change the type of an existing object.

With our model, where types are a consequence of classification, we do not have any of these problems. An existing object can be inserted in a new collection and thus gain the attributes that correspond to this collection or it can be deleted from a collection and lose the corresponding attributes. An object can be in many collections at the same time, so it can have many *types* at the same time.

All three forms of object evolution are possible, and even easy to implement within our model. In fact, object evolution is reduced to the simple *insert* and *delete* collection operations. Evolution is here so simple because the model that we present here has been developed from the very start with this idea of object-evolution in mind. Actually, the concept of *typing by classification* together with the *multiple simultaneous classification* is really the cornerstone of our model, and this is precisely what makes object evolution so easy and so elegant.

4.2 Schema Evolution

The second form of evolution that our model must support is the *schema evolution*. This form of evolution affects only the collections and the associations of a database system, but the problem here is that the impact on the other objects and on the consistency of the database (see 6) can be considerable. The following list enumerates the following mutations that can occur in schema evolution:

- Adding a collection
- Removing a collection
- Changing the attributes of a collection
- Changing the ISA relationship between collections
- Changing the source or the destination of an association

In the rest of this section, we will see the consequences of each of these mutations. Let us start with the *insertion* of a new collection into an existing database system. This change has no impact on the consistency of the rest of the system and just consists in creating a new collection-object and defining the attribute definitions bounded to that collection. In our example of the university database, this simple evolution form may occur, for example, when we need to add a new classification like *golf-player* which will contain all the persons of the university who play golf. We could then use this classification to store the “handicap” of each golf player.

The second mutation that we want to describe in this section is the *deletion* of a collection. This operation is not more complex than the previous one, provided that the collection that we want to delete is *empty*. Actually, if the collection being deleted is not empty, all the objects that are in this collection would be in an inconsistent state. The solution is either to automatically delete all the members of a collection that is deleted or to only allow the deletion of a collection if it is empty.

The next mutation we have to deal with is the *change of attributes definition*. This mutation can be either the *insertion* of a new attribute definition, the *suppression* of an attribute definition, the *change of the type* of an attribute definition or the *change of the name* of an attribute definition. This last change, namely the change of the name is the simplest one because it has no influence on the values of the attributes. Unfortunately, this is not the case with the other three changes; in these more complicated cases, the system (or the user) needs also to update all members of the collection that has been changed. We will see in 9 how this problem can be solved.

Another mutation is the change of the classification hierarchy by changing the ISA relationship between collections. This mutation should only be needed in some very rare situations, but in order to be complete, our model also has to support this operation. In our example, this could happen if we decide to add collections like *Golf Player*, *Horse Rider*, etc. and to consider them as a sub-collection of a new collection like *Sportsmen/Sportswomen*. This mutation seems to be quite complex, but in our implementation, it is straightforward: actually, it only requires to change the sub/super-collection informations between the collections. This case will also be discussed in more detail in 9.

The last mutation that could happen is the change of the source or the destination of an association. Like the previous one, this mutation is often due to bad initial design and should only occur in some very rare cases. The problem here is that we have to check that all pairs, members of the association, are still valid in the new schema. This problem is similar to the one we had with the change of attributes definition and we choose the same method for solving it.

In the two preceding sections, we have explained what evolution is and how our model supports this concept. We have shown that, thanks to the fusion of classification and types, simple schema evolution is easy to do and that even complex schema evolution is still possible, although not as easy. All this has been possible because evolution, and more precisely object evolution, has been sought from the very beginning of the conception of the model.

Chapter 5

Query and Algebra

In the preceding chapters, we explained the structure of our model, the way to insert and to delete objects from collections and the concept of the evolution of the objects, but it still misses a fundamental feature of a DBMS, namely a mechanism for *retrieving* the information that we stored in a database. This task is done by a so called *query processor* and this chapter describes the one that applies to our model. The section 5.2 also gives a formal specification of the algebra that rules our query processor.

5.1 Query Processing

The mechanism for processing queries in almost all DBMSs is similar to the one shown in figure 5.1. The picture is divided in three columns which represent the three main actions of the processing.

In the leftmost column, we have the *front-end* of the processor, which task consists in translating a query from a *human readable* form into a *computer readable* form. The human readable form is expressed in a so called *query language* and is usually in a textual representation of the query. The computer readable form is called the *intermediate representation* and has generally the form of a graph or a tree.

The middle column shows the *optimizer*. Its role is to try to *tune* the intermediate representation in order to make the evaluation of the query as fast as possible. Query optimization is a complex topic and many people are working on it. As our focus is more on the classification model than on a very fast query processor, we will only give an overview of how such an

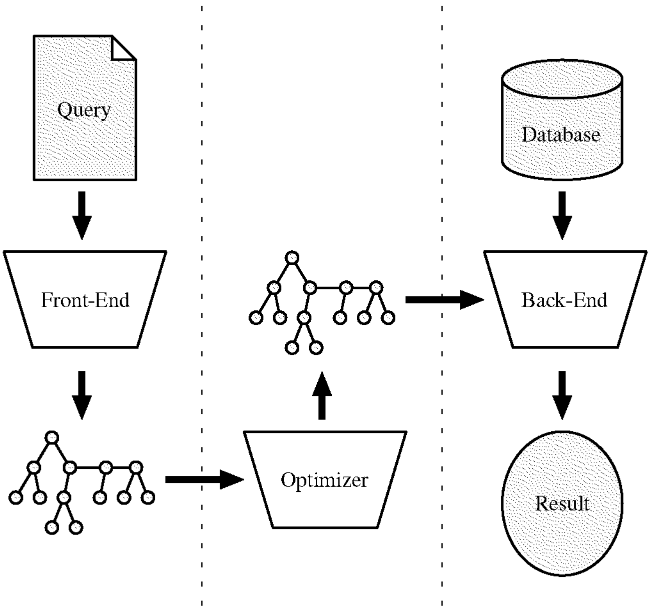

Figure 5.1: Query Processing

optimizer could be used in our system, and we will only implement some simple optimizations to prove the feasibility of the concept.

The last operation is the evaluation of the query and is illustrated in the rightmost column of the figure. The task of this operation, called *back-end*, is to read the intermediate representation of the query and to *filter* the database with respect to the instructions given in this intermediate representation. The result is then a new collection that contains only the objects that passed through the filter specified by the query.

A very important characteristic of our query model is that it is restricted to its primary goal, that is to evaluate queries and to *extract* informations from a database. This is different from some popular query languages, like SQL, where the same language is used to define the schema of the database, to express queries and also to make insertions, updates and deletions in the database. With our model, the result of a query is always a collection, and no existing object is ever changed during the evaluation of a query. However, as our system is designed to be integrated in a programming environment, it is always possible to write a procedure or a function to *post-process* the result of a query in order to extract some computed information. Let us take our example of a university again and suppose that we want to do some statistical computations over the marks of all students of the course on “Compiler Construction”. The first operation consists in extracting all students who attend the “Compiler Construction” course and once we have our result, we can then write a small procedure that scans the resulting collection and compute the statistical values that are searched.

5.2 The Algebra

The *algebra* of a model is the set of *operations* that are supported by this model. This section describes the algebra, based on OM [Nor92], that rules our model and its query mechanism.

5.2.1 Operations on Collections

All operations on collections take one or two collections as operand and return a new collection as resultant of the operation. As this resultant is also a collection, it has to be inserted somewhere in the schema hierarchy, or in other words, we have to find a *father* for that resultant. The simplest solution would be to say that all resultants are *orphans*, that is that they do not have any father, but we can be a little bit smarter and try to give a suitable father to the resultant. In order to do so, we need to define the concept of the *least common supercollection*, but before defining what this is, let us first define what a *common supercollection* is: We say that S is a common supercollection of two collections X and Y, if S is a supercollection of the X and S is a supercollection of the Y. In a more formal way, we can write that S is a common supercollection of X and Y if $X \leq_c S$ and $Y \leq_c S$.

Having defined what a common supercollection is, we can now extend this definition to explain what is a *least* common supercollection: L is the last common supercollection of two collections X and Y if L is a common supercollection of X and Y and if there does not exist any other common supercollection S of X and Y that is also a subcollection of L. In a formal style, L is the least common supercollection of X and Y if $X \leq_c L$, $Y \leq_c L$ and $\nexists S \mid X \leq_c S, Y \leq_c S$ and $S \leq_c L$. We write this as: "$L = X \sqcup Y$". If X and Y have no common supercollection, then $X \sqcup Y$ is the pseudo collection $\mathbb{U}$ that represents the universe, which is the pseudo-collection of all objects from a database system.

Having defined the concept of the least common supercollection, we can now continue the description of the algebra by considering first the operations that apply to *every* collection. Let us start with the three basic operations *union*, *intersection* and *difference* borrowed from discrete mathematics.

Union

The result of the *union* between two collections C_1 and C_2 is a collection R that contains all the elements that are in C_1 *or* in C_2. Further, the direct supercollection of R is the least common supercollection of C_1 and C_2. In a formal form, this gives:

$$R = C_1 \cup C_2 \Rightarrow \begin{cases} ext(R) = \{o \mid o \in ext(C_1) \vee o \in ext(C_2)\} \\ \uparrow R = C_1 \sqcup C_2 \end{cases}$$

Remark: This operation is commutative and associative, so $C_1 \cup C_2 = C_2 \cup C_1$ and $C_1 \cup C_2 \cup C_3 = (C_1 \cup C_2) \cup C_3 = C_1 \cup (C_2 \cup C_3)$.

Intersection

The result of the *Intersection* between two collections X and Y is a collection R that contains only the elements that are in X *and* in Y. As in the union operation, the direct supercollection of R is the least common supercollection of X and Y:

$$R = X \cap Y \Rightarrow \begin{cases} \mathsf{ext}(R) = \{o \mid o \in \mathsf{ext}(X) \wedge o \in \mathsf{ext}(Y)\} \\ \uparrow R = X \sqcup Y \end{cases}$$

Remark: Like the union operation, the intersection is also commutative and associative.

Difference

The result of the *Difference* between two collections X and Y is a collection R that contains only the elements that are in X *and not* in Y. Here also, the direct supercollection of R is the least common supercollection of X and Y:

$$R = X - Y \Rightarrow \begin{cases} \mathsf{ext}(R) = \{o \mid o \in \mathsf{ext}(X) \wedge o \notin \mathsf{ext}(Y)\} \\ \uparrow R = X \sqcup Y \end{cases}$$

Remark: Contrary to the two preceding operation, the difference is neither commutative nor associative.

The two following operations have no direct relation with discrete mathematics, but are rather inspired from standard query languages like SQL [DD97] or from other object oriented database management systems.

Selection

The first operation in this group is the *selection*. The goal of this operation is to extract (or to select) some objects from a given collection X. The criterion to decide if an object will be in the resulting collection or not is called a *filter* and is expressed by a boolean function $\mathcal{F}$ that takes an object as parameter and returns *TRUE* if the object is accepted by the filter or *FALSE* if it is rejected. The result of the selection is a new collection, which is a subset of X. The direct supercollection of the result is thus the collection X itself.

$$R = X\%\mathcal{F} \Rightarrow \begin{cases} \mathsf{ext}(R) = \{o \mid o \in \mathsf{ext}(X) \wedge f(o) = \mathsf{TRUE}\} \\ \uparrow R = X \end{cases}$$

We will see in the second part of this dissertation that, for efficiency reasons, we decided to implement different variants of the selection operation, but in any case, the goal of these variants is the same, that is the extraction of a subcollection from a given collection by using a filter.

Flattening

The second and last operation of this group is the *flattening*. This function is used to transform nested collections into flat collections. This operation is only defined on collection X of collections (nested collection) and its result is then a new collection R. As the objects in R have been flattened, they are no more in the same context as before, so the direct supercollection of R has no relationship with the one of X and is defined as the pseudo-collection $\mathbb{U}$.

$$R = \pm X \Rightarrow \begin{cases} \mathrm{ext}(R) = \{o \mid \exists N \in X \wedge o \in N\} \\ \uparrow R = \mathbb{U} \end{cases}$$

5.2.2 Operation on Associations

The operations that we have described up to here are defined for all type of collections, but there are also some other operations that are only defined for a special sort of collection, namely for the associations. We begin this group of operations by those who extract the source or the destination of an association:

Domain Selection

The first operation in this group is the *Domain Selection* and its goal is to build a collection with all the objects x from the *source* of an association A for which exists an object y in the destination of A and where $\langle x, y\rangle$ is a member of A. As this collection R is a subcollection of the source of A, the direct supercollection of R is then the source of A.

$$R = \mathrm{Dom}(A) \Rightarrow \begin{cases} \mathrm{ext}(R) = \{x \mid \exists y \wedge \langle x, y\rangle \in A\} \\ \uparrow R = \overleftarrow{A} \end{cases}$$

Range Selection

Similar to the domain selection, the *Range Selection* builds a collection with all objects y from the destination of an association A for which exists an object x in the source of A and where $\langle x, y\rangle$ is a member of A. The direct supercollection of R is the destination of A.

$$R = \mathrm{Rng}(A) \Rightarrow \begin{cases} \mathrm{ext}(R) = \{y \mid \exists x \wedge \langle x, y\rangle \in A\} \\ \uparrow R = \overrightarrow{A} \end{cases}$$

The next four operations are somehow similar to a *selection* operation in the sense that the result R of these operations is a subcollection of the operand A, but unlike in the selection, the filter here is not given by a *custom* function, but by a collection C.

Domain Restriction

The first operation in this group is the *Domain Restriction*. As we said before, the result R of this operation is a subcollection of the first operand A, and the filter returns *TRUE* if, and only if the object x from the pair $\langle x, y\rangle$ from A is a member of the collection C. As R is a subcollection of A it is natural to define that the direct supercollection of R is A itself.

$$R = A \text{ dr } C \Rightarrow \begin{cases} \mathrm{ext}(R) = \{\langle x,y\rangle \mid \langle x,y\rangle \in A \wedge x \in C\} \\ \uparrow R = A \end{cases}$$

Domain Subtraction

The *Domain Subtraction* is similar to the domain restriction and the only difference is that here, the filter returns *TRUE* if, and only if the object x from the pair $\langle x, y\rangle$ from A is *not* a member of the collection C.

$$R = A \text{ ds } C \Rightarrow \begin{cases} \mathrm{ext}(R) = \{\langle x,y\rangle \mid \langle x,y\rangle \in A \wedge x \notin C\} \\ \uparrow R = A \end{cases}$$

Range Restriction

In the *Range Restriction*, the filter returns *TRUE* if, and only if, the object y from the pair $\langle x, y\rangle$ from A is a member of the collection C.

$$R = A \text{ rr } C \Rightarrow \begin{cases} \mathrm{ext}(R) = \{\langle x,y\rangle \mid \langle x,y\rangle \in A \wedge y \in C\} \\ \uparrow R = A \end{cases}$$

Range Subtraction

The last operation of this group is the *Range Subtraction* and the difference between this operation and the previous one is that in the range subtraction, the filter returns *TRUE* if, and only if, the object y from the pair $\langle x, y\rangle$ from A is *not* a member of the collection C.

$$R = A \text{ rs } C \Rightarrow \begin{cases} \text{ext}(R) = \{\langle x, y\rangle \mid \langle x, y\rangle \in A \wedge y \notin C\} \\ \uparrow R = A \end{cases}$$

The last group of operations that are only defined for associations are the following ones:

Inversion

The *Inversion*, as its name says, is used to invert an association. In the result R, the source becomes the destination of the operand A, and its destination becomes the source of A. As the result of the inversion does not necessarily keep the context of the association, the direct supercollection of the result is the pseudo-collection $\mathbb{U}$.

$$R = \text{inv}(A) \Rightarrow \begin{cases} \text{ext}(R) = \{\langle x, y\rangle \mid \langle y, x\rangle \in A\} \\ \uparrow R = \mathbb{U} \end{cases}$$

Composition

The *Composition* of associations is similar to the *composition of relations* of the discrete mathematics. This operation allows the use of a sequence of two associations A_1 and A_2 to define a new collection R. The context of this new association R has no relation with either operand and thus, the direct supercollection of R is the pseudo-collection $\mathbb{U}$.

$$R = A_1 \circ A_2 \Rightarrow \begin{cases} \text{ext}(R) = \{\langle x, z\rangle \mid \\ \quad \exists y \wedge \langle x, y\rangle \in A_1 \wedge \langle y, z\rangle \in A_2\} \\ \uparrow R = \mathbb{U} \end{cases}$$

Remark: This operation is not commutative, so $A_1 \circ A_2$ is not necessary the same as $A_2 \circ A_1$. However, the composition is associative, so $A_1 \circ A_2 \circ A_3 = (A_1 \circ A_2) \circ A_3 = A_1 \circ (A_2 \circ A_3)$.

Nesting

The *Nesting* operation is a sort of grouping operation in that for each object x in the domain of the operand A, it forms a pair consisting of that object x and the collection of objects of the range of A. Here also, the result R is not necessary in the same context of the operand A, so the direct superclass of R is the pseudo-collection $\mathbb{U}$.

$$R = \mathrm{nest}(A) \Rightarrow \begin{cases} \mathrm{ext}(R) = \big\{\langle x, C\rangle \mid \\ \quad x \in \mathrm{dom}(A) \wedge C = \{y \mid \langle x, y\rangle \in A\}\big\} \\ \uparrow R = \mathbb{U} \end{cases}$$

Unnesting

Unnesting is the opposite of the previous operation and it expands an association A which contains pairs comprising a collection C as a destination.

$$R = \mathrm{unnest}(A) \Rightarrow \begin{cases} \mathrm{ext}(R) = \{\langle x, y\rangle \mid \langle x, C\rangle \in A \wedge y \in C\} \\ \uparrow R = \mathbb{U} \end{cases}$$

Division

The *Division* operation takes an association A and a collection C and returns the collection R of all objects x from the domain of A such that the object x is linked to every member of the given collection C. Once again, the result is in a completely new context and the direct supercollection of R is $\mathbb{U}$

$$R = A \ \mathrm{div}\ C \Rightarrow \begin{cases} \mathrm{ext}(R) = \{x \mid \forall y \in C \wedge \langle x, y\rangle \in A\} \\ \uparrow R = \mathbb{U} \end{cases}$$

Closure

The last operation of our algebra is the *Closure*. The result R of this operation is the reflexive transitive closure of the association A. Here again, our definition is similar to the one of discrete mathematics. The direct supercollection of the result of a composition is $\mathbb{U}$, so the result of the closure is also the pseudo-collection $\mathbb{U}$.

$$R = *A \Rightarrow \begin{cases} \mathrm{ext}(R) = \mathrm{ext}\left(\bigcup_{i=0}^{\infty} C^i\right) \\ \quad \text{where} \quad \mathrm{ext}(C^0) = \mathrm{ext}(id_C) \\ \qquad\qquad\qquad\quad\ = \{\langle x, x\rangle \mid \exists y \wedge \langle x, y\rangle \in C\} \\ \quad \text{and} \quad \mathrm{ext}(C^1) = \mathrm{ext}(C) \\ \quad \text{and} \quad \mathrm{ext}(C^i) = \mathrm{ext}(C^{i-1} \circ C) \\ \uparrow R = \mathbb{U} \end{cases}$$

In this chapter we gave an overview of the query processing and a detailed description of the algebra supported by our model. In the second part of this dissertation, we will come back to this subject and show how this has been implemented in our system Odéon.

Chapter 6

Constraints

As we have described it in the previous chapters, our model of classification is used to give a *structure* to a database but, until now, we have not discussed how to keep the informations within this structure in a consistent state. For example, in a database about persons, how could we prevent a person from being at the same time in a collection *Men* and in a collection *Women*? Of course, one could argue that this problem is under the responsibility of the application that manages the database, but as we want to always *guarantee* the integrity of the database system, we preferred to integrate a mechanism into the system and not leave this responsibility to the application. This chapter explains how, in our model, we can guarantee the integrity of a database system by using the so called *constraints*.

6.1 Transactions

Before we go into the details of the constraints and their specifications, we want to make clear *when* the database must be consistent and *when* it could be in an inconsistent state. Actually, it would be desirable to just say that the database must *always* be consistent, but unfortunately, this is not possible and we want to illustrate this with an example: in our university database, we have a collection of persons, which is then subdivided in a collection of students, a collection of assistants, and a collection of professors. Suppose now that in this database, we defined a constraint that guarantees these three points:

- No object can be at the same time in more than one collection

- If an object is in the collection of persons, it must also be in one of the collection of students, assistants, or professors.
- If an object is in one of the collection of students, assistants, or professors, it must also be in the collection of persons.

Now suppose that a student becomes an assistant. How can we update our database without violating the constraints at any time? We can try to first delete the object from the collection of students and then to insert it in the collection of assistants, or on the contrary, to first insert the object in the collection of assistants and then to delete it from the collection of students, or we can even try to first delete the object from the collection of persons, but in fact, we will never be able to *always* keep the database within the constraint. The obvious way to circumvent this problem is to group a number of operation together in a so called *transaction*, and to consider this group in an atomic way.

Such a transaction occurs during a given period of time; it always starts by a *begin transaction* and ends either by a *commit transaction* or by an *abort transaction*. The semantic of a transaction is defined like this:

If a database is consistent before the beginning of a transaction, then the system guarantees that the database will also be consistent after the end of a transaction.

As we have just seen, the transactions are required by our model to preserve consistency, even if our system is designed as a single-user system. However, as we know that our system will be a single-user system, we can slightly simplify our transaction mechanism by dropping the begin transaction operation. Actually, if there is only one user, it is obvious that after he ends a transaction, he will either start a new one or close the database he is working on. So, our transaction model works like this:

- When a database is opened, a new transaction is automatically started.
- When a transaction is aborted, the system recovers its state from before the begin of transaction and a new transaction is then started.
- When a transaction is committed, and after having checked all the constraints, the system updates the database and starts a new transaction as well.
- At the end, when the user closes the database, the current transaction is aborted, and the database is effectively closed.

Until now, we only considered the role of transactions for guaranteeing the *logical* integrity of a database system, but transactions do not have to restrict them-self to this role. Actually, they can also be used to guarantee the *physical* integrity of a database system. This role is not part of the logical model, but we will come back to this issue in the second part of this dissertation.

6.2 Classification Constraints

In our model, the logical consistency of a database system is defined by a set of rules called *constraints*. A constraint is an invariant that must be checked and validated at the end of each transaction, in order to allow its commitment. We do not have to check the constraints at the beginning of a transaction, because as we have seen in the previous section, we already *know* that the constraints have to be valid before the beginning of a new transaction.

In our model, the consistency constraints are specified by an equation between two algebraic expressions. A constraint is said to be *valid* only if both sides of the equation are equal, otherwise we say that the constraint is *violated*. The formal definition of the constraint equations is given by the following syntax:

constraint = expr "=" expr.
expr = term {"∪" term}.
term = factor {"∩" factor}.
factor = "("expr")" | collection | ∅.

For example, to impose that the collection *Men* and *Women* must be disjoint, we write the following constraint:

$$\text{Men} \cap \text{Women} = \emptyset$$

Some models, like OM, have a *higher* language for defining the constraints, but we will show that our simple model is powerful enough to express all the constraints that OM can define, and even more if we want. The rest of this section describes all the constraints from the OM model and their equivalent within our model.

Isa Constraint

In OM, the isa constraint is noted "$C_1 \preceq C_2$" and means that every object classified as C_1 is also classified as C_2. This subject has already been studied in chapter 3 and is already automatically checked by our type system, but in order to be complete, we still want to give an equivalent constraint with our constraint algebra. This constraint could simply be expressed like this:

$$C_1 \cup C_2 = C_2$$

or

$$C_1 \cap C_2 = C_1$$

Disjoint Constraint

The disjoint constraint is noted "$C \leftarrow C_1 \mid C_2$" in OM, and means that no object classified as C can be classified as both C_1 and C_2, where C_1 and C_2 are both subcollections of C (in our model, the disjoint constraint, as well as the following ones, is not restricted to the case where C_1 and C_2 are subcollections of C and we leave the responsibility to the user of our model to determine if it makes sense to set a constraint on such unrelated collections). With our algebra, this constraint is simply expressed by the following equation:

$$C_1 \cap C_2 = \emptyset$$

Cover Constraint

The cover constraint is noted " $C \leftarrow C_1 + C_2$" and means that every object from the collection C must be also classified as C_1 or C_2 (or both), where C_1 and C_2 are (usually) subcollections of C. In our constraint algebra, an equivalent formulation is given by:

$$C_1 \cup C_2 = C$$

Partition Constraint

The partition constraint is in fact a combination of the two preceding constraints. It is noted "$C \leftarrow C_1 \ddagger C_2$" and means that every object classified as C must also be classified as exactly one of C_1 or C_2, where C_1 and C_2 are like before, (usually) subcollections of C. The equivalent formulation with our algebra is a set of the two equations of the two preceding constraints:

$$\begin{cases} C_1 \cap C_2 = \emptyset \\ C_1 \cup C_2 = C \end{cases}$$

Intersection Constraint

The last constraint defined in OM is the intersection constraint. It is noted "C_1 & $C_2 \leftarrow C$" and means that every object classified as both C_1 and C_2 must also be classified as C. In OM, it is also specified that C is a subcollection of C_1 and C_2, but as our model does not support multiple inheritance, we extend this constraint to the case where C_1 and C_2 do not have special properties. With our algebra, this constraint is simply expressed by the following equation:

$$C_1 \cap C_2 = C$$

In OM, the constraints are not restricted to be defined on two collections and it is possible to define, for example, that the collections C_1, C_2 and C_3 have to be a partition of a forth collection C. It is not a problem for our model to express these constraints as well. For the previous example, an equivalent formulation in our system would be:

$$\begin{cases} C_1 \cap C_2 = \emptyset \\ C_2 \cap C_3 = \emptyset \\ C_1 \cap C_3 = \emptyset \\ C_1 \cup C_2 \cup C_3 = C \end{cases}$$

In this section, we have shown that the algebra that we defined for our model is very simple to use, simple to check and expressive enough to represent all usual constraints. We will see in the second part of this dissertation how these constraints have been efficiently implemented in our system.

6.3 Object Constraints

With the type of constraint that we described in the preceding section, it is now possible to check the logical consistency at the *collection* level, but it would also be useful to check the logical consistency at another level, namely at the *object* level. By object level, we mean the object with its *attributes*, and a constraint at the object level is used to check the validity of these attributes in a given context. To illustrate such a constraint, one could define a constraint to restrain the age of the members of a collection *Employees* to be a number between sixteen and seventy-five.

Contrary to the constraints that we explained in the previous section, it is no more possible here to define the object constraints with a simple equation or even a set of equations. In fact, we really want to allow here every imaginable constraint and the most obvious way of allowing that, is to define the constraint using a general purpose programming language. This is exactly what we did, and while our system is already designed to integrated in a programming environment, we just choose the language of this environment to express our object constraints. In the implementation that we will show in the second part of this dissertation, the host language is Oberon, thus our object constraints will be specified by an Oberon function that takes an object as argument and returns a boolean value indicating whether or not the object is conform to the constraint.

As with the classification constraints, the object constraints are also checked at the end of the transactions. This works in the following way: During a transaction, every object that is modified or inserted into a collection is marked. Then, at the end of the transaction, the validation functions of all marked objects are called and the product of all results is checked. If the product is FALSE, this means that at least one of the validation function returns FALSE, so the transaction will fail to commit. On the contrary, if the product is TRUE, this means that all object constraints are valid and the transaction could then commit.

6.4 Cardinality Constraints

The last type of constraint that our model supports is the cardinality constraint and applies only to associations. The idea behind this constraint is not new and was already specified in 1976 in the well known *Entity-Relationship* model [EN94, Chapter 3].

The goal of cardinality constraints is to verify that neither too many nor too few objects are related through a given association. In our model, as in many other data models, such a constraint is given by two pairs of numbers: $(m : n \rightarrow o : p)$. The first pair $(m : n)$ specifies that every member of the collection at the source of the association must participate in at least m and at most n instances of the relationship represented by the association. If n is replaced by a star (“$*$”), this means that there is no maximum limit. The second pair $(o : p)$ is similar to the first one, but for the collection at

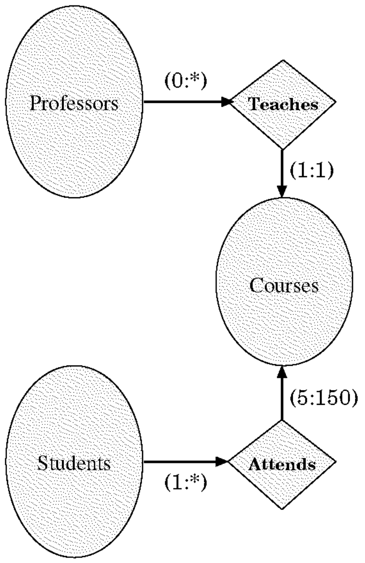

Figure 6.1: The Associations in the University Database

the destination side of the association.

Let us take our example of the university database to illustrate this type of constraint. In this example, we have collections of *students*, of *professors* and of *courses*. A professor teaches courses and a student attends them. This gives us the two associations *teaches* and *attends*. A pair from the *teaches* association is valid if it satisfies the following constraint: *a course must be taught by one, and only one professor*. In this example, a professor is not obliged to teach a course and he can teach as many courses as he want, so there is no further constraint for this association. On the other hand, a link from the *attends* association is valid if it satisfies the following constraint: *A student must at least attend one course* (and he can attend as many as he want). On the other hand, a course only exists *if there is at least five students who attend it*, and as class rooms have only place for a hundred and fifty students, *a course can not be attended by more than a hundred and fifty students*. The schema of such a database as well as the cardinality constraints is given by the figure 6.1

This constraint is specific enough to deserve a special treatment in the implementation. If we do it in an appropriate way, we can check these constraints in a more efficient way than if it were to be checked using "generic" constraint mechanism. This issue will be explained in details in part II.

6.5 Propagation

In the preceding sections, we specified how to define constraints, but we did not explain *what* happens when a constraint can not be satisfied at the end of a transaction. We already know that in this situation, we are not allowed to commit the transaction, but then, what can we do? Two obvious solutions are possible: either the transaction is simply rejected and the database is *reset* to the same state as before the transaction, or the system tries to *restore* the constraint by updating the database and, only when the system is not able to do this in an *unequivocal* way, the transaction is rejected. In our system, we chose the second approach, and we achieved this by using the concept of *constraint propagation*.

Let us show in an example how a smart update propagation mechanism works and what it does when a constraint is violated: suppose, for example, that the simple constraint $Men \cup Women = Persons$ is defined in a database and that, at the end of a transaction, the system detects that an object has been added to the collection *Women*, but is not in the collection *Persons*. The constraint is thus violated, but our smart system can restore it in an unequivocal way by propagating the insertion of the object to the collection *Persons* as well. Actually, there would be a second way to restore the constraint, that would consist in *undoing* the change by removing the object that we just inserted in the collection *Women*. However, this second way does not make much sense, so we can still claim that there is an unequivocal way to restore the transaction if we add *without undoing the change that the user has explicitly made*.

Update propagation could be very difficult to achieve in some OODBMSs, but with our system, our simple constraint model makes this operation straightforward. Here is how it works: first of all, we have to represent the constraints in a convenient way (for the computer), and the most suitable one is still by using binary trees. The root of the tree is the "equal" symbol of the constraint and the other nodes, are either collections (collection nodes) or one of the two operations *union* or *intersection* (operation node). The figure 6.2 shows a graphical representation of such a tree for the constraint "$A \cup B = C$". As we can see in the picture, every node, except for the root, has a *father* node. An operation node always has two *sons*, and the two sons of a same father are called *brothers*.

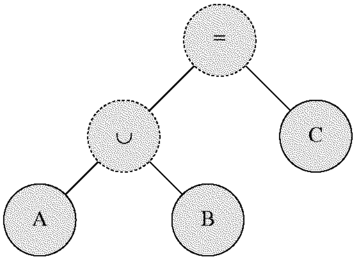

Figure 6.2: The Tree Representing a Constraint

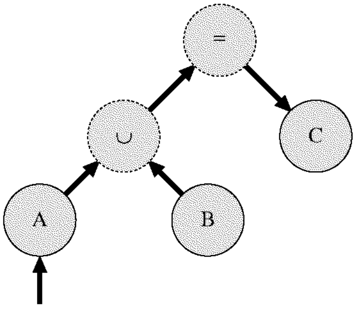

Figure 6.3: First Pass of the propagation

When an object is inserted or deleted from a collection, we consider that this operation comes from the *bottom* of the collection (Figure 6.3). The update is then propagated bottom-up through the operation nodes until it reaches the root of the tree. From there the update is propagated top-down along the other branch of the tree (Figure 6.3). When the update reaches a leave, it turns once again, and propagates once again bottom-up toward the root of the tree. The process continues like that until no more change occurs.

To understand the whole process, we still have to explain how propagations are done through the operation nodes. In order to do that, let us first enumerate all the different actions that can occur:

- An operation node can be an intersection or a union
- The update can be an insertion or a deletion
- The update can come from the bottom or from the top

UPDATE	OP.	DIR.	ACTION
Ins	$\cup$	Up	If the brother does *not* already have the object that is inserted, then the update is propagated to the father.
Ins	$\cup$	Down	This case can not be resolved in an unequivocal way!
Ins	$\cap$	Up	If the brother already has the object that is inserted, then the update is propagated to the father.
Ins	$\cap$	Down	The update is propagated to both sons.
Del	$\cup$	Up	If the brother does *not* have the object that is deleted, then the update is propagated to the father.
Del	$\cup$	Down	The update is propagated to both sons.
Del	$\cap$	Up	If the brother has the object that is deleted, then the update is propagated to the father.
Del	$\cap$	Down	This case can not be resolved in an unequivocal way!

Table 6.1: Update Propagation in Operation Nodes

So all in all, there are eight different actions that can occur at the level of an operation node. The table 6.1 summarizes all these situations.

We can summarize this chapter by saying that constraints play an important role in our model because they ensure that our databases are consistent. It is true that consistency is not the only aspect that a DBMS must take care of, but how useful is a very fast DBMS if the information contained in the database could not be kept consistent? In fact, we considered that consistency was so important that we decided to build this feature in the system itself. The model that we present here uses constraint to ensure the consistency. These constraints are simple to express and can be checked in an efficient way. On top of that, our model uses the concept of update propagation to resolve many constraint violations and to save many pointless operations to the user.

Implementation

Chapter 7

The Persistent Store

The persistent store is an essential building block of the system, because it is the place where the objects are stored.

Instead of extending the Oberon heap with a virtual memory facility and a paging/swapping mechanism, we decided to implement an independent *Persistent Store Manager* (PSM) tailored and optimized for our specific needs. This section describes the structure of the persistent store and explains how the PSM manages it.

7.1 Features of the Persistent Store

Before going into the details of the implementation, let us explain what we should expect from such a persistent store.

persistence The first requirement is, of course, that the storage is persistent, this means that the storage keeps its data even in case of power failure. There are many persistent media, but for our purpose, magnetic disks are still the best choice for their capacity/speed/price ratio. With modern disks, the persistence of the data is guaranteed over many decades and the reliability can even be enhanced by using techniques such as disk shadowing [BG88] or RAIDs* [CLG$^+$94].

efficiency In addition to being persistent, our store has to be fast. We just said that we were using a magnetic disk as physical medium and its

*Redundant Arrays of Independent Disks

maximum speed will be the limiting factor. With a modern disk, the average time to access and transfer an 8K block between the disk and memory is about 10ms. These values are physical limits, and the only way to make disks faster is to use them in parallel, like in RAIDs. Using five disks will reduce the costs to about 2ms but this is still a lot in comparison to memory-to-memory transfer time. The solution is then to use caching techniques to reduce disk transfer (see Section 7.7). As we exactly know the structure of the objects that are going to be stored, and as we are designing a persistent store for supporting our model only, we have the opportunity to make the store very efficient.

reliability The last requirement is that the storage must be reliable. We have already seen that magnetic disks provide a good physical reliability, but we also provide logical reliability, that is that we have to ensure that the store is always in a consistent state, and that, if a failure occurs, the system is able to recover a consistent state with a minimum of data loss. This point was also decisive for the choice of the store structure.

7.2 The Organization of the Store

In order to fulfill the requirements given in the previous section, and to harmoniously support our model, we decided to structure the persistent store in the same way as the Log-Structured File Systems [OD89], [RO92].

The main idea of the *log-principle*, in the context of an object store, is to consider the store as a *tape*. There are two heads on the tape, namely a reading head and a writing head. The reading head can be freely positioned on the tape, whereas the writing head can only append data at the end of what is already written on the tape (Figure 7.1).

With this technique, an object is never changed *in place* but always completely rewritten at a new place. This gives the ability to the system to be very robust because, as we never delete or overwrite objects, every operation can be undone. Thus, the system can always recover by restoring a previous *good* status. This is used to implement transactions and to recover in the case of physical failures such as a power failure.

Besides the reliability of the storage, the log-structure also provides excellent support for our evolution model. We said in the first part that, when an object evolves, it may also gain or lose some attributes, so actually,

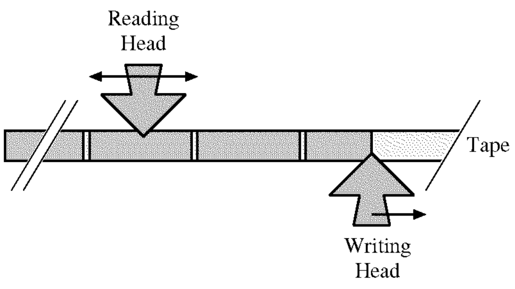

Figure 7.1: The Log-Principle

the *size* of the object may change because of the evolution. With the log-structure, this is not a problem because nothing requires that the updated object is of the same size as the original. The unavoidable drawback of this flexibility is that the address of an object, that is its physical location, will not be constant. Consequently, in order to find a given object, we need a method to *translate* an object ID into the physical location of that object. In our implementation, this translation is done using an *ID table*. When an object moves, the corresponding entry in the table is modified with the new location of the object. Another drawback of this model is the fact that the data written to the store continuously grows. So we may end with a store filled with old versions of objects. We avoid this problem by implementing a *packer* which role is to collect the garbage left by old versions and to compress the database. The *packer* is described in Section 7.8 of this chapter.

To summarize, the persistent store consists of two main data structures: the log-structure for storing the objects themselves and the OID table for storing the location of the object. As both structures are growing, we decided to start the log-structure at the beginning of the store, growing toward its end, and to start the OID table at the end of the store, growing toward its beginning (Figure 7.2).

7.3 Streams

In the previous section, we used a tape to explain the working of the log-structure. However, we have also seen that, for numerous reasons, a magnetic disk would be the ideal storage medium for our data. To bridge the

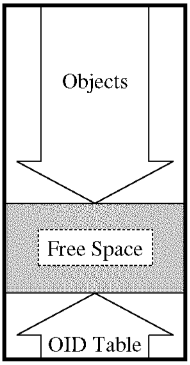

Figure 7.2: The Simplified Structure of the Store

two concepts, we decided to implement our system using the abstraction of *streams*.

The concept of streams is not a new concept; in 1971, J. E. Stoy and C. Strachey have already implemented streams as first class citizen in their OS6 [SS72] experimental operating system for small computers. The concept has then been implemented, among others, in Unix [RT74] and in Xerox's Pilot Operating system [RDH$^+$80] written in Mesa [MMS79]. Today, the concept is still widely used in modern systems like Java. The version 4 of the Oberon system does not use the concept of streams but it uses *files* which are conceptually the same. In our system, we could have extended the existing files, but as we wanted to change the existing system as little as possible, we decided to implement the streams as a new, separate structure.

In all these systems, streams are data structures used to transfer sequential data from a sub-system to another. In 1971, streams have been used for communications between the central unit and the terminal or between the central unit and the tape reader. Today, streams are used to exchange date with magnetic discs, between two computers over a network, ...

In order to describe our implementation of streams, we will compare them with a tape, so we will speak of *head* position to describe the position where data is written or read. However, this is just an abstraction; our concept of streams is not limited to tapes, and as we will see later, it is very generic and could also apply to disks, network, etc.

The streams implemented in our system support the following operations:

- We can set its head to a given position:
  PROCEDURE (s: Stream) **SetPos***(pos: LONGINT)
- We can query the current head position:
  PROCEDURE (s: Stream) **Pos***(): LONGINT
- We can read a raw blok of n bytes from the current head position:
  PROCEDURE (s: Stream) **ReadBytes***(VAR x: ARRAY OF SYSTEM.BYTE; n: LONGINT)
- And we can write raw a blok of n bytes from the current head position:
  PROCEDURE (s: Stream) **WriteBytes***(VAR x: ARRAY OF SYSTEM.BYTE; n: LONGINT)

In addition to raw transfer, streams also allow for the transfer of formatted information. This is even one of the most important feature of streams. In our implementation we provide the following methods:

```
PROCEDURE (s: Stream) Read*(VAR x: SYSTEM.BYTE)
PROCEDURE (s: Stream) ReadBool* (VAR x: BOOLEAN)
PROCEDURE (s: Stream) ReadInt* (VAR x: INTEGER)
PROCEDURE (s: Stream) ReadSInt* (VAR x: SHORTINT)
PROCEDURE (s: Stream) ReadLInt* (VAR x: LONGINT)
PROCEDURE (s: Stream) ReadSet* (VAR x: SET)
PROCEDURE (s: Stream) ReadReal* (VAR x: REAL)
PROCEDURE (s: Stream) ReadLReal* (VAR x: LONGREAL)
PROCEDURE (s: Stream) ReadString* (VAR x: ARRAY OF
  CHAR)
PROCEDURE (s: Stream) ReadNum* (VAR x: LONGINT)
PROCEDURE (s: Stream) Write*(x: SYSTEM.BYTE)
PROCEDURE (s: Stream) WriteBool* (x: BOOLEAN)
PROCEDURE (s: Stream) WriteInt* (x: INTEGER)
PROCEDURE (s: Stream) WriteSInt* (x: SHORTINT)
PROCEDURE (s: Stream) WriteLInt* (x: LONGINT)
PROCEDURE (s: Stream) WriteSet* (x: SET)
PROCEDURE (s: Stream) WriteReal* (x: REAL)
PROCEDURE (s: Stream) WriteLReal* (x: LONGREAL)
PROCEDURE (s: Stream) WriteString* (VAR x: ARRAY OF
  CHAR)
PROCEDURE (s: Stream) WriteNum* (x: LONGINT)
```

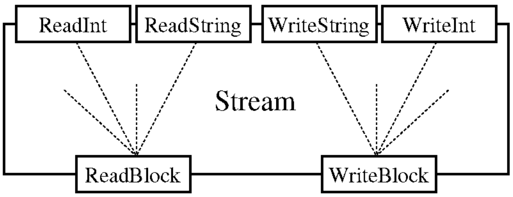

Figure 7.3: Generic Stream

All methods are self-explanatory, apart from the **ReadNum/ WriteNum**. Actually, these two methods are used to *encode* and *decode* LONGINTs in a more compact representation. **WriteLInt** will always write four bytes, whereas **WriteNum** will write a compressed representation of a LONGINT that needs one to five bytes, depending on the value of the argument.

These stream methods rest on two generic transfer methods, namely:

```
PROCEDURE (s: Stream) ReadBlock*(adr, len: LONGINT;
   VAR data: ARRAY OF SYSTEM.BYTE)
PROCEDURE (s: Stream) WriteBlock*(adr, len: LONGINT;
   VAR data: ARRAY OF SYSTEM.BYTE)
```

These two methods are *abstract*, this means that they are only used to *define* the methods and not to *implement* them(Figure 7.3). In order to use streams, it is necessary to define an *extension* of the type Stream and in the extension, to implement both **ReadBlock** and **WriteBlock** methods. In our implementation, we decided to use disks for storing the information, so we implemented *disk streams* (Figure 7.4). This means that the stream is mapped to raw sectors of a physical disk. However, is is possible to extend streams for other media such as a network (Figure 7.5).

The Odéon system is not limited to disk-streams; any sort of streams can be used with the system, but in our system, we have implemented streams only for disks.

7.4 The Disk Streams

We decided to use disk-streams in our system for optimal performance. A disk-stream directly accesses the sectors of the disk, without going through

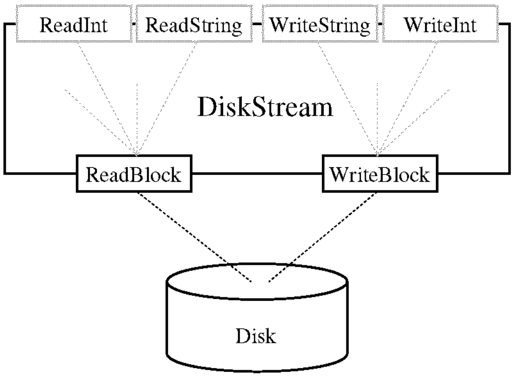

Figure 7.4: Disk Stream

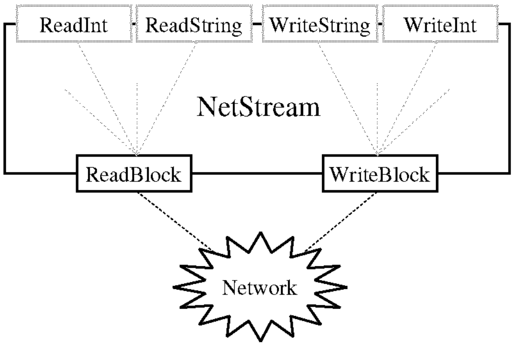

Figure 7.5: Network Stream

the overhead of a file system. This makes the storage very efficient but requires a dedicated disk and is hardly portable. For this reason, we decided to encapsulate all the low-level disk access procedures in a single module, namely the "DiskIO" module. Beside the initialization of the disk and the computation of access statistics, "DiskIO" basically provides the two following procedures for transferring raw sectors from and to the disk:

```
DiskIO.ReadRawSectors(sector, len: LONGINT;
   VAR buffer: ARRAY OF SYSTEM.BYTE);
DiskIO.WriteRawSectors(sector, len: LONGINT;
   VAR buffer: ARRAY OF SYSTEM.BYTE)
```

The arguments for these two procedures are

sector the first sector of the disk that is transfered

len the number of sectors that are transfered

buffer the memory blok that is read or written during the transfer

On top of the *DiskIO* module, we have the *Disk* module. This module is *portable* and uses only the services provided by the *DiskIO* module. The role of the *Disk* module is first to provide a "low-level" caching mechanism at the level of disk sectors. The cache that we implemented is very simple, with the classical LRU (Least Recently Used) replacement algorithm [HP90, Section 8.3]. The second role of the *Disk* module is to hide the sector structure by allowing the transfer of blocks of any size, starting at any position. This will then make the implementation of disk-streams straightforward.

The data are read from and written to the disk by the following procedures:

```
PROCEDURE ReadBlock(adr, len: LONGINT;
   VAR data: ARRAY OF SYSTEM.BYTE);
PROCEDURE WriteBlock(adr, len: LONGINT;
   VAR data: ARRAY OF SYSTEM.BYTE)
```

We implemented the *write-back* cache mechanism, i.e. when the data is written to the stream, it stays for a while in the cache and is not directly written to the stream. This method is much faster than the *copy-through* cache mechanism, where the stream is directly updated, but is also more hazardous because of the delay between the writing to the cache and the writing to the disk. To restrict the risk of data loss, the system calls regularly the FlushCache procedure, which writes unsaved data to the stream. A cache flush can also be forced by directely calling the procedure FlushCache.

Beside the opening and the closing of the disk, the implementation of the DiskStreams module is reduced to

```
MODULE DiskStreams;
  TYPE
    Stream* = POINTER TO StreamDesc;
    StreamDesc* = RECORD (Streams.Stream)
  END;


  PROCEDURE (s: Stream) ReadBlock*(adr, len: LONGINT;
    VAR data: ARRAY OF S.BYTE);
  BEGIN
    Disk.ReadBlock(adr, len, data)
  END ReadBlock;


  PROCEDURE (s: Stream) WriteBlock*(adr, len: LONGINT;
    VAR data: ARRAY OF S.BYTE);
  BEGIN
    Disk.WriteBlock(adr, len, data)
  END WriteBlock
END DiskStreams.
```

7.5 The Log-Structure

We have seen in Section 7.2 that the store was divided in three main parts: the objects themselves are stored in the log-structure at the beginning of the store, the OID table is stored at the end of the store, and there is a third region between these two parts that is the free space. In addition to these three main dynamic parts, there is a fixed size header at the very beginning of the store. Figure 7.6 shows in detail the structure of persistent store.

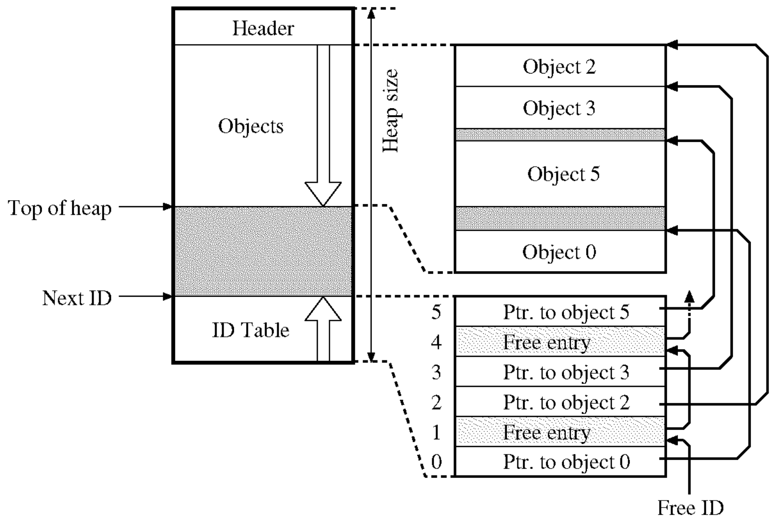

Figure 7.6: The Detailed Structure of Persistent Store on Disk

7.5.1 The Header

Let us first describe the header of the persistent store. The header is a critical part of the store because it contains the size of each part (objects, free and OID table) and without this information, the persistent store is unusable. We will see later that the objects themselves contain some redundant information to allow the reconstruction of the header (and of the OID table). Let us see exactly what information is written in the header:

Signature Four bytes used to identify the memory space. In the current implementation, the signature is the string ”PSF” terminated by the 0X character.

Info The data stored in this field is not relevant for the system. It is just a small text for the human only. In the current version, this field contains the string “PSF Version 1.0”.

Heap size This is the size of the whole persistent store memory.

Free ID Pointer to the beginning of the free ID list.

Next ID Pointer to the first unallocated ID.

Top of heap Address where the next object will be written.

7.5.2 The OID Table

The role of the OID table is to provide fast access to an object, given its OID. In fact, this table is a *hash table*, in which the indices are the OID themselves. Because these OIDs are unique, we have no collision in the OID table and this is why it can be so fast. In the table, we store the address of the corresponding object, or in other words, the offset from the start of the stream where the object is stored.

Since the OID table is *at the end* of the store, the OID is actually subtracted from the size of the store to give the entry. The exact formula is:

$$\text{Entry Location} := \text{Heap Size} - (\text{OID} + 1) * \text{Address Size}$$

When the system has to allocate a *new* object, it first looks for a free OID. This can be either a new OID or a recycled one, that is an OID that has been formerly used and is now again available. This situation occurs when an object is completely removed from the store, that is removed from all collections. In order to find all recycled OIDs, these are linked together and rooted by the *Free ID* field of the header.

The OID always points to the last version of the object. This implies that all previous versions of an object are lost, unless we extend the object itself with a pointer to its previous version. This would be easy to do and so we could easily add versioning to our system. However, in the current implementation, the system can only access the last version of an object.

7.5.3 The Objects

The object described here is the father of every object that needs persistence. In other words, every persistent object is an extension (direct or indirect) of the object described here. To avoid confusion, the object described here is called the *PSM-object*. In our system, an object can be in two different states: *awake* or *sleeping*. An object is *awake* if it is loaded in the heap of the system and can be used as usual Oberon object. An object is *sleeping* when it is removed from the heap and thus only stored on the stream. We will see later how an object changes from the *awake* status to the *sleeping* one and vice-versa, but for the moment, let us examine the form of an object loaded in memory.

Each PSM-object has following the attributes:

id the OID of the object

state This field contains information that is mainly used by the *garbage collector* and the *packer*.

dTriggers When an object is deleted from the store, the system may need to do some cleaning actions in order to keep the database in a consistent state. Such an action is called a *deletion trigger*. As there may be more than one trigger, *dTriggers* are in fact an array of triggers and each of them is called when the object is deleted. We implemented the triggers as pointers to procedures, allowing *up-calls* [RW92, Section 11.3.5] of procedures. We used this technique instead of adding a new method to the object because sometimes (see Section 7.9 for an example), we just need to add a trigger to one single object and not to a whole class of objects.

Besides attributes, each PSM-object implements the following methods:

Externalize This method writes to a stream all the attributes and all information needed to reconstruct the object [Szy92], [Tem94]. For PSM-objects, this method only writes the names of the deletion triggers, but we will see later how extensions of PSM-objects such as O-objects or collections use this method to write their attributes. As the stream is a *serial* data structure, this externalization process is sometimes also called *serialization*.

Internalize This method is used to read data back from a stream and to re-build the object. It is the opposite of the *Externalize* method.

Init Init is a virtual method that is called when a new object is allocated or after it has been internalized.

AllocateObject This method allocates a valid OID to the object. It also calls its Init method and puts the object in the cache.

AllocateSystemObject Similar to the previous method, AllocateSystem allocates an OID to the object, but here, the OID is given as parameter. Actually, the *Allocate* method will never allocate OIDs less than 256, so objects with OIDs between 0 and 255 are what we call *System-objects* and have to be allocated with this method. A typical example of such a system-object is the *root-table* which needs to be directly accessed by a predefined OID.

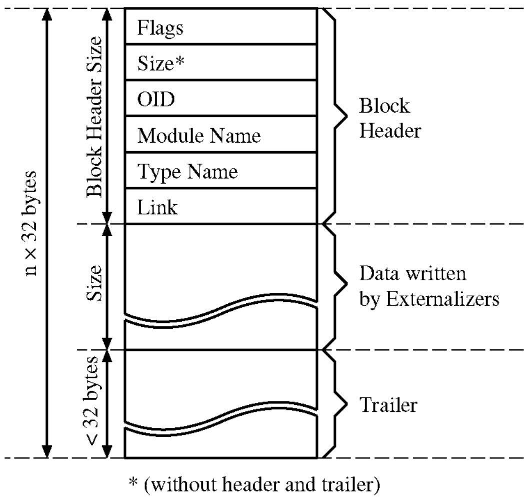

Figure 7.7: Structure of Sleeping Object

Touch This last method sets the *dirty* bit of the object. In other words, this method tells the object that it has been changed and that it needs to be written back (or externalized) to the stream.

When an object is *sleeping*, its data is only stored in the stream. The structure of such a sleeping object is shown in Figure 7.7. In the figure, we distinguish three main blocks: the header, the data written by the externalizers and the trailer.

In the header, we store the following information:

Flags These are some flags used by the system, more precisely by the *garbage collector* and the *packer*

Size This is the size of what is written by the externalizers and not the total size of the block on the stream. However, the total size can easily be computed: blockSize := blockHeaderSize + size + (−blockHeaderSize − size) MOD 32

OID This is the OID of the object. This information is not directly needed by the system because it always uses the OID table to access the objects, but if, for one reason or another, the OID table is corrupted, then the system will use this field to reconstruct the table.

ModuleName Our persistent store must not only store PSM-objects, but also extensions of them. In order to be able to recreate the object in the heap, we need to know which extension it is and therefore we need to know in which module the extension is defined.

TypeName We have seen how to store the name of the module where the extension is defined; now we still need to store the name of the type itself. This field is for that. We used arrays of characters to represent the *ModuleName* and the *TypeName*, but a more economical solution would be to manage a *type table*, where all *ModuleName/TypeName* would be stored and to only store an *index* to this table in the object. This solution would make more compact objects, but it has two serious drawbacks: first of all, the object would have to rely on an additional structure and would no longer be self-contained and second, the *type table* would be a dangerous weak-point of the system. If the *type table* is damaged, then *all* objects become unusable. For these reasons, we decided to use the full type names in each object.

Link This last field is used by the *packer* and will be described later.

7.6 The Persistent Store Manager

The role of the persistent store manager is to *synchronize* the persistent store and the Oberon-heap. This consists of two main tasks:

- To allocate new objects in the persistent store and to remove the ones that are no longer reachable.
- To awake objects that are needed by the application and to put to sleep those that are no longer used.

7.6.1 Objects Allocation

To *allocate* an object means to *register* it to the persistent store by assigning it its OID. This OID is either a new one, i.e. an OID that has never been used until now, or it is a recycled one, i.e. the OID of an object that has been *deallocated*.

7.6.2 Objects Deallocation

When an object is deallocated, this means that it is definitely removed from the persistent store and its OID can by recycled. This operation is very dangerous because, before the deallocation of an object, we need to be sure that this object is no longer referenced by other objects. We have to ensure, for example, that the object is not in any collection anymore. Instead of giving this responsibility to the user, we preferred to give it to the system by implementing a garbage collector. The user will never explicetly delete an object, but he will either unregister it from the root table, or remove it from a collection. When an object is neither in the root table, nor member of any collection, it is no longer reachable by the system and becomes a candidate for being removed by the garbage collector.

The ideas behind our garbage collector are quite simple and work according to the *mark and sweep* principle.

For that system to work, each object needs a special attribute, namely the *marked* attribute. This attribute is set during the *mark* phase of the garbage collection and tells if a given object is reachable or not. This has been implemented in the following way:

1. All objects are first scanned and each one is unmarked.
2. We call the *Mark* method of all *system* objects, that is of all objects that have a predefined OID. Each object is then responsible for marking itself as well as all other objects that are reachable from it. As the root-table is a system object, all objects that are registered in the root table will be registered and thus all reachable objects will be marked.
3. This last step is the *sweep* phase. All objects are scanned once again, and the ones that are not marked (those that are not reachable) are deallocated.

The deallocation of objects by the way of a garbage collector is very convenient but it also has a serious drawback: it has to lock the database during the garbage collection and this may take a very long time. Therefore if the system needs to be permanently available, we must either renounce the use of garbage collection and deallocate objects manually, or implement an incremental garbage collector.

7.6.3 Waking up Objects

Waking up objects means loading objects from the persistent store to the heap and making them available for the application. The point here is that if the application wants to load an object that is already loaded, the system should not reload the object again, but it should return the reference of the already loaded object. In order to do that, the system needs to know which objects are already loaded and which are not. This could be implemented with a hash table, but because this table will be subject to many insertions and deletions, we preferred to implement it as a balanced tree instead of a hash table. This balanced tree is called *loadedObjects* and is implemented as a *Symmetric Binary B-Tree* [Bay72], [Wir86], also called *red-black trees*.

Here is the pseudo-code of the *LoadObject* procedure of the PSM:

```
PROCEDURE LoadObject*(id: LONGINT;
  VAR o: SYSTEM.PTR);
BEGIN
  IF id is valid THEN
    Look for o in LoadedObject tree
    IF not found THEN
      Internalize object o from stream;
      Insert o in LoadedObjects tree
    END
  ELSE o := NIL
  END
END LoadObject;
```

7.6.4 Putting Objects to Sleep

As long as the application keeps pointers to the object, the object will stay in the heap, but when the object is no longer referenced, then it should be removed from the heap, otherwise the heap would be quickly filled with garbage. Fortunately, the Oberon system already has a garbage collector for objects in the heap and we should be able to use it for our objects as well. However, two problems prevent us from using it directly:

- The objects are still reachable, via the *loadedObjects* tree, so they will still be marked.

- If the object has been changed, it must be externalized to the stream before being removed from the heap.

The solution for the first problem is to use *weak* pointers [Szy92], [Atk89] for the pointers to the objects in the *loadedObjects* tree. The garbage collector of Oberon traces only *strong (or regular)* pointers, and thus, does not see the loaded objects if they are not referenced by some other *strong* pointers. There is no direct support for *weak* pointers in Oberon, but we achieve the same result by using the type *LONGINT* instead of *POINTER* and converting them with *SYSTEM.VAL* when they need to be dereferenced. This solution is not the most elegant one, but is the only one that is possible without changing the Oberon system. Besides that, this solution is very simple and very efficient.

The second problem is more complex to solve and we cannot solve it without extending the Oberon system. Here we need a procedure or a method that would be called just before the object is removed. Such a method is called a *Finalizer* [Szy92] and in our case, the finalizer will check if the object is *dirty*, i.e. if the object in the heap has been changed in comparison with the object in the stream, and if so, the object will be externalized to the stream. The finalization mechanism has to be implemented at the same level as the Oberon garbage collector, so we have to implement it in the *kernel* of Oberon.

The *Kernel* has been extended with two new procedures, namely:

```
PROCEDURE RegisterObject*(obj:SYSTEM.PTR;
  finalize:Finalizer);
PROCEDURE ReleaseObject*(obj:SYSTEM.PTR);
```

where the type *Finalizer* is defined as:

```
PROCEDURE(obj:SYSTEM.PTR).
```

The procedure *RegisterObject* links the finalizer *finalize* with the object *obj* and ensures that the finalizer will be called before the object is removed and the procedure *UnregisterObject* unlinks the object *obj* and its finalizer.

The concept of finalizer is useful, not only for a persistent store, but also for many object-oriented applications. It allows, for example, to cleanly close a file or a network connection when an object that has an handle to the file or the network connection in question is deleted. Besides that, it is very easy to use; we just have to call a procedure of the kernel and the object is registered with its finalizer. Even the implementation of finalizers seems to be straightforward, but the problem is that we have no control about what the finalizer does. A finalizer could, for example, define a strong pointer to its object and then the object could no longer be deleted. We do not want to go too deep in the details of the implementation of the finalizers here because this subject has already been discussed in [Szy92] and [Tem94].

7.7 Caching Mechanism

We already have a simple cache mechanism at the level of disk sectors, but within the persistent store manager, we also have the possibility to include a cache mechanism at the level of objects. The only thing we have to do is to establish some *strong* pointers pointing to the objects we want in the cache and so we ensure that they will not be removed by the garbage collector.

```
CONST cacheSize = 32;
VAR cache: ARRAY cacheSize OF RECORD
   info: LONGINT; obj: Object
END;
```

In this datastructure, *info* is used for the replacement algorithm and *obj* is the strong pointer to the cached object. In the current implementation, *info* is used as a timestamp and the replacement algorithm is the simple *Least Recently Used* algorithm. This “high-level” cache is more efficient than the one described in Section 7.4, but it also has a serious drawback: It is very hard to control the size of the cache. When we cache disk sectors, we know exactly the size of a sector and we know the number of sectors that we want to cache, so the total size of the sector-cache is easy to compute. But when we cache objects, we can hardly know the size of an object and thus, we cannot know the total size of the object-cache. For this reason, our cache contains only few objects (32 in the current implementation).

7.8 Packer

The log-structure is a very convenient way of storing objects, but it also uses memory in a very extravagant way. This is only acceptable if we need to keep *all* the different versions of all objects, but this is seldom required. Usually, we are only interested in the last version of an object, and all the previous ones can be deleted. However, deleting an object does not help reducing the size of the storage because new objects will still be written at the end of the stream. The deleted objects only make holes in the stream, but they do not serve to retrieve memory. In order to solve this problem, we implemented a *packer*. Its role is to reduce the size of streams by reorganizing the objects and filling the holes.

The packer can work in two different ways: *at-once* or *incremental*. In the first variant, the database is locked during the whole process and is thus not available for any application. This works in the following way:

1. Each object of the stream is scanned and if the object is a last version, then it is linked together with the other last version objects.
2. All the objects in the list are rewritten one after the other, without holes in between. The corresponding entries in the OID table are also updated to reflect the change.

If the system requires 100% availability, we cannot afford to lock the database for long and we have to adopt the second variant of the packer, namely the *incremental* one. In this variant, the packing process is split into many small operations, and the system is only locked during the small time required to do a single operation. All these different operations are done by the *PackStep* procedure which is regularly called by the simple tasking system of Oberon.

The incremental packing process is divided into four main phases, namely *initializing*, *scanning*, *compacting* and *ending*. The incremental packer stores its current phase in the *packState* variable, so when *PackStep* is called, it knows its current phase and thus knows what to do. Let us now see in detail what happens in the different states.

1. Initializing (Figure 7.8): in this phase, the incremental packer makes a copy of the *Top of heap* pointer and initializes the pointers *packAdr* and *packEnd* for the next phase.
2. Scanning (Figure 7.9): in this state, each time that the *PackStep* procedure is called, one additional object is scanned and linked with the previous ones. The pointer *packAdr* is incremented each time by the size of the object until it reaches the end of the heap represented by the pointer *packEnd*
3. Compacting (Figure 7.10): When the last object has been scanned, the incremental packer changes to the *compacting* phase. In this phase,

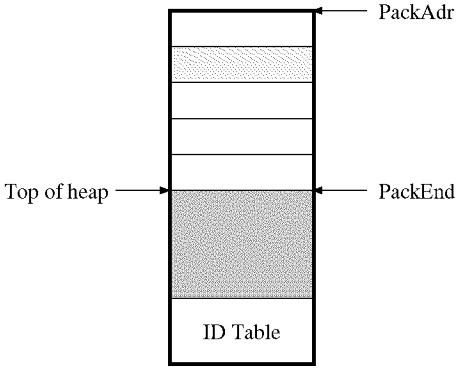

Figure 7.8: Incremental Packer - Initializing

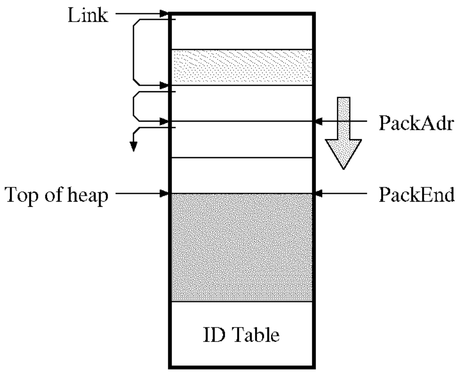

Figure 7.9: Incremental Packer - Scanning

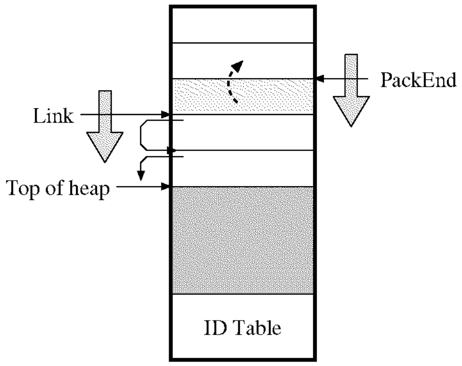

Figure 7.10: Incremental Packer - Compacting

the linked list of objects is followed and each object in the list is moved, if possible, towards the beginning of the heap. At the same time, the corresponding entry of the OID table is updated to reflect the change in the heap. As in the previous phase, this operation is not atomic, but only a single object is handled each time that the procedure *PackStep* is called. During this phase, the variable *packEnd* points to the end of the already compacted region of the heap.

4. Ending (Figure 7.11): When the linked list of objects has been fully traversed, the incremental packer enters the *ending* phase. Here the system checks if objects have been written to the heap during the packing process. If not, then the *top of stack* pointer is updated and the incremental packer returns to the the *initializing* state. However, if some objects have been written during the process, then the store is still in an inconsistent state (because the last block is not necessarily a valid free block) and it must start again, considering the new objects as well. So if the packer notices that the actual *top of stack* is not the same as the one it stored in the *initialization* phase, then it sets the variables *packAdr* and *packEnd* again, makes a copy of the *Top of heap* pointer and returns to the *scanning* phase.

The incremental packer, as described in this section, has been implemented in our system as an Oberon Task. It will thus only be active between Oberon commands. So if a command is running for a long time, making a lot of changes in the database, it should, now and then, explicitly call the incre-

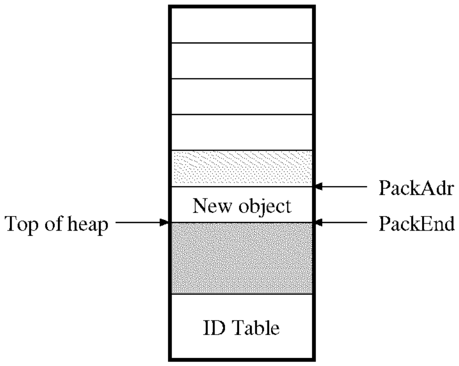

Figure 7.11: Incremental Packer - Ending

mental packer. The incremental packer has been designed to also work when the system is in a transaction.

We still have to consider a final aspect of the packer, namely the robustness of the packer. If a hardware failure occurs during the *initializing* or the *scanning* phase of the packer, there is no consequence at all, but if the failure occurs in the middle of a *compacting* step or during the *ending* phase the store may be left in an inconsistent state. We cannot avoid this problem without complex algorithms. In the current implementation, we decided not to handle this particular problem, so our system requires that each packing step can be done in an atomic way.

7.9 The Root Table

We have already seen that the *root table* is some sort of directory, which allows access to objects by a name instead of their OID. In this section, we will present the details of the implementation of the root table management.

First of all, we have to define precisely the basic requirements of the root table management. This table is used to store a set of pairs <Name, OID>, and we must be able to *add* new pairs, to *delete* pairs and to *look for* a given pair.

For efficiency reasons, we decided to implement the root table using B-Trees [BM72], [Com79]. This structure will also be used for maintaining secondary indexes so we decided to implement a *generic* B-Tree data structure. This data structure and the corresponding algorithms are implemented in the module *GPBT (Generic Persistent B-Tree)*. We do not want to show here the details of the GPBT, but rather the way we used it to implement the root table. Let us start with the definition of the Root object:

```
TYPE Root = POINTER TO RootDesc;
  RootDesc = RECORD (PSM.Object)
  directory, invDir: GPBT.Tree
END;
```

We see that the root is an object with two additional system attributes, namely *directory* and *invDir*, both of type *GPBT.Tree*. The attribute *directory* is a pointer to a B-Tree where all pairs <Name, OID> are stored. This tree is indexed by *Name*. The *invDir* is also a pointer to a B-Tree where all pairs are stored, but this one is indexed by *OID*.

Why do we need this second B-Tree? In what cases do we need to access a pair by its *OID* instead of by its *Name*? In fact, there are two reasons for the use of the *invDir* tree:

1. To detect and to avoid *aliases*.
2. To delete the corresponding directory entry when an object is deleted.

The first reason is obvious: we want to prevent objects from being registered more than once in the root table because aliases would cause confusions. If an object is registered in the root table, it is registered under a single name.

The second reason is also clear: in order to keep the consistency of the system, we cannot allow an entry of the root table to represent an object that is not in the store. So when an object is deleted, we must check the root table to see if this object is registered, and if yes, then we have to remove it from the table as well. This is achieved using the *deletion triggers* that have been explained before.

The basic functionality of the root table is implemented by these three procedures:

```
PROCEDURE AddRoot* (o: PSM.Object;
  name: ARRAY OF CHAR);
PROCEDURE DeleteRoot* (name: ARRAY OF CHAR);
PROCEDURE ThisRoot* (name: ARRAY OF CHAR):
  PSM.Object;
```

To allow enumeration off all the entries of the root table, we also implemented this procedure:

```
PROCEDURE EnumRoot* (enumerator: RootEnumerator);
```

Finally, we also have a procedure for initializing the root table, and another to just open it for further operations:

```
PROCEDURE MakeSystemRoot*;
PROCEDURE OpenSystemRoot*;
```

Chapter 8

Transactions

In traditional database systems, transactions [GR92] are primarily used to allow simultaneous accesses to the database by several users (or several processes) and for recovery from failures. As our system is a single user system, we do not need to deal with the problem of simultaneous accesses, however, as we have seen in Chapter 6, we still need a form of transaction to correctly implement our model of constraints.

In our system, a transaction is rather a form of *validation*. We want to ensure that, at a given moment, the database is in a consistent state regarding all its constraints.

8.1 The computation of the validity envelope

During a transaction, all objects that are changed are registered in the *PSM.update* list. When we want to commit the transaction, we begin by computing the set of all objects that have to be checked. This set is called the *validity envelope* and is obtained by calling the *AddToValidityEnvelope* method of all objects of the *PSM.update* list. This *AddToValidityEnvelope* method checks if the object is already in the envelope, and if not, it adds it and calls the *AddToValidityEnvelope* of all related objects. Let us illustrate this method with the description of the procedure that adds a DB-Object to the validity envelope:

```
PROCEDURE (o: Object) AddToValidityEnvelope*;
   VAR i, len: LONGINT;
```

```
BEGIN
   IF ~o.InValidityEnvelope() THEN
      o.AddToValidityEnvelope↑;
      IF o.belongsTo = NIL THEN len := 0
      ELSE len := LEN(o.belongsTo↑) END;
      i := 0; WHILE i < len DO
         o.belongsTo[i].collection.AddToValidityEnvelope; INC(i)
      END;
   END;
END AddToValidityEnvelope;
```

In this procedure, we see that if the object is not already in the envelope, then it is first added to it, and then, all the *AddToValidityEnvelope* methods of all collections to which the object belongs are recursively called.

The procedure that we just presented is only valid for one form of object, namely the DB-Objects. Other forms of objects like Collections or Associations have other *AddToValidityEnvelope* methods. If the user adds a new object type to the system, then he also has to provide a corresponding *AddToValidityEnvelope* method.

The computation of the validity envelope is a sensitive part of the system because if we forget some objects, then the database may, in spite of everything, end in an inconsistent state. On the other hand, if we take too many objects, the validation takes too much time. Because this point is very sensitive, our system also provides a way to check all objects of the database. However, this operation may need a lot of time and should only be used as a debugging tool for finding implementation errors.

8.2 Validation

Once that we have the validity envelope, we still need to check or to *validate* each object of the envelope. This is achieved by calling the *Validate* methods to each object of the envelope. This *Validate* method checks the validity of the object and stores the result in the state attribute of the object.

When all objects have been checked and if all are valid, then we assume that the system is consistent and we finalize each object by calling its *Commit* method. After we have committed a transaction, we want to be sure that the *PSM.updates* list is empty, i.e. that all modified objects have been finalized. As with the *AddToValidityEnvelope* method, the *commit* methods are bound to the form of object. Here as well, we may have user defined methods that are not under our control. Because it is not impossible (but rather improbable) that a given *commit* method changes some other objects, we repeat the validation of the *PSM.updates* list until the list is empty.

The following procedure (in pseudo-code) shows the implementation of the commit transaction operation:

```
PROCEDURE CommitTransaction*;
BEGIN
   REPEAT
      SBB.Init(validityEnvelope);
      SBB.Enumerate(updates, ComputeValidityEnvelope);
      SBB.Enumerate(validityEnvelope, ValidateObject);
      validTest := TRUE;
      SBB.Enumerate(validityEnvelope, CheckValidity);
      SBB.Init(validityEnvelope);
      IF validTest THEN
         IF flush # NIL THEN flush END;
         tmp := updates; SBB.Init(updates);
         SBB.Enumerate(tmp, CommitObject);
      ELSE
         HALT(Exceptions.OdeonTrap);
      END
   UNTIL SBB.First(updates) = NIL
END CommitTransaction
```

As we have seen in the preceding pseudo-code, if the transaction does not produce a valid state, en exception is raised. In Oberon, this can be done using the *HALT* instruction. In the original Oberon V4, the *HALT* instruction produces an interrupt handled by the Kernel of Oberon by interrupting the current command and by displaying the state of the system in a so called *Trap Viewer*.

Together with our system, we extended the Oberon Kernel by giving to the programmer the possibility to catch some of these exceptions and to handle them in another way. In our system, the exception *Exceptions.OdeonTrap* is caught by the Odéonsystem and does the following action:

1. The current transaction is aborted, that is, *PSM.update* list is cleared and all objects are reset to their state after the last successful *commit*.
2. A viewer is opened on the display to explain that an error occurred.

PSM.update This is the default handler for *Exceptions.OdeonTrap* exceptions. In a real application program, the programmer can implement his own exception handler which would replace the default one. In his own handler, the programmer can do whatever he wants: he could try to correct the problem and try to commit the transaction again, or he could abort the transaction.

With this mechanism, we give a lot of freedom to the application programmer.

As we have seen, the validation of the constraints of the database is a very sensitive point of the system: we have to ensure the validity of the complete database by checking as few objects as possible. We have shown that by using the *updates* list and the *validity envelopes* it is possible to achieve such a goal and our implementation is in fact very efficient. With our system, we have shown that it is possible to combine a general constraint mechanism together with an efficient consistency checking algorithm.

Chapter 9

Objects and Collections

We have seen in the preceding chapter how the persistent objects are implemented, but until now, we still have objects with only system attributes, i.e. objects whose attributes are defined at compile time, in the system itself. In order to support our model of *typing by classification*, we need to implement a higher level object class as well as a collection class. This chapter explains how these classes have been implemented in Oberon and how they are used by an application.

9.1 Collections

It is not easy to decide to begin this chapter with the definition of collections or with the definition of objects because on one hand, collections *are* themselves objects, and on the other hand, an important part of objects, namely their attributes, are defined by collections. We found that the second point was more important and so just keep in mind the fact that collections are objects too and let us begin this chapter with the description of collections.

As we will see later, our system needs two sorts of collections: *persistent* collections and *transient* ones. The former store the persistent information and the latter are used as temporary collections in our query mechanism. In order to represent the two different sorts of collections, we decided to define a common type, namely the type *Container*:

```
TYPE
   Container* = POINTER TO ContainerDesc;
```

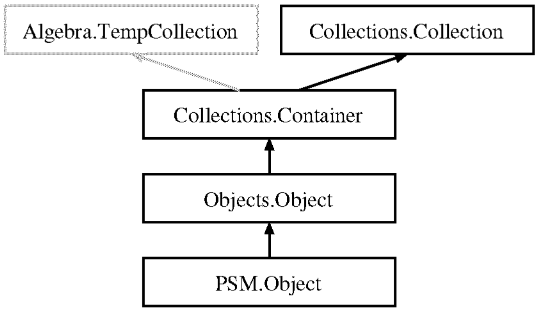

Figure 9.1: Type Hierarchie below the Type Collection

```
ContainerDesc* = RECORD (Objects.ObjectDesc);
   father*: Collection;
   ext*: Objects.List
END
```

In addition to the attributes of the type *Object*, the *Container* has two system attributes: the *father*, used to represent the *ISA* relationship between collections, and the *ext* field, for storing the members of the collection. As we can see from the definition, the type of the *ext* field is a *list*. In fact, in order to allow faster searching, *ext* is implemented as a binary tree, but in order to keep things simple, we can just consider it as a list.

The type *Collection* is used to define persistent collections. Figure 9.1 summarizes the type hierarchy that we just explained, and the rest of this sub-section shows* how Collections are implemented:

```
CollectionDesc* = RECORD (ContainerDesc);
   sons: IdList;
   checkers: CheckerList;
   attrDefs: AttributeDefList;
   propagations: ColNodeList;
   associations: CollectionList;
   indexes-: IndexList;
   ins, del: Objects.TList
END
```

*The fields in gray will be explained later

The first field, the *sons*, is the counterpart of the *father* of a container. In our model, we do not support multiple inheritance, thus a container has at most one father and this can be implemented by a simple pointer. It is not so easy to represent the sons of a collection because the collection can have more than one son and thus cannot be represented by a single pointer. As this information does not change so often, we decided to implement it as a dynamic array. Another fact is that, for most of the operations, the father does not need its sons, so we decided to implement the sons as an array of OIDs instead of an array of pointers. The advantage of this solution is that when a collection is loaded from the persistent store into the heap, the system does not also automatically load all its sons. The drawback is then that if the system needs the son of a collection, it has to explicitly load it, but as we said, this is rarely needed. In fact, the only case where the system needs to access the sons of a collection is when an object is removed from a collection. As we will see later, our system does *update propagation*, so it will propagate the deletion and will also recursively remove the object in all sons of the collection.

The next field that we want to describe here is *attrDefs*. This field is used to store the attributes defined by the collection and is of primary importance for our model of typing by classification. This field is of type:

```
AttributeDefList = POINTER TO ARRAY OF AttributeDef
```

where AttributeDef is defined as:

```
AttributeDef = RECORD
  name: Objects.Name;
  type: Types.Type
END
```

The field attrDefs is a POINTER TO ARRAY, or in other words, a *dynamic array* of attribute definitions and an attribute is defined by a Name and a Type.

Once again, we decided to use a dynamic array instead of a list or a tree because usually, there are only few attributes defined by a collection, they do not change often and as they are accessed often, they need to be accessed fast. Therefore, the dynamic array seems the best implementation possible. We will see later how objects use this structure to acquire their attributes.

The last fields that we want to describe here are ins and del. These two fields store the changes made to the content of the collection during a transaction. This allows us to efficiently check the constraints and, if necessary, to *undo* the changes made to a collection. Hence, these fields are relevant only during a transaction.

When an object is inserted or deleted from a collection we first update the *ext* field. Then we also report the change in the fields ins and del, where we store all insertions and deletions. If the transaction is aborted, we use these two sets to restore the ext field to its original state. On the other hand, if the transaction is committed, then we just clear both ins and del fields.

9.2 Objects

In our system, the Objects are the entities that carry the information. We have already seen that this information is represented by the attributes of the objects and that these attributes are defined by the collections to which the object belongs. We have already seen in the previous section how collections are implemented, so we can now continue with the implementation of the objects. In Odéon, the type *Object* is defined as follows:

```
Object* = POINTER TO ObjectDesc;
ObjectDesc* = RECORD(PSM.ObjectDesc)
  handler: ObjectHandler;
  belongsTo-: POINTER TO ARRAY OF Membership
END
```

We see that the type *Object* is an extension of the type *PSM.Object* (that we called PSM-Objects). To avoid confusion, we will call these objects *DB-Object*.

In other words, we can say that the *PSM.Objects* are the objects at the persistent store level, that is, an object with an *OID* and all other attributes that we have described in Section 7.5.3. On the other hand, *DB-Objects* are more elaborate than *PSM.Objects*. They are used to implement the objects of our data model, that is, to implement objects with attributes belonging to collections.

DB-objetcs are implemented as an extension of PSM-Objects, so they inherit all the properties of PSM-Objects. In addition to those properties, the DB-Object type defines a *handler* and a *belongsTo* attribute. Let us start with the description of the header:

```
ObjectHandler = PROCEDURE (o: Object; VAR msg: ObjectMes-
sage)
```

This *handler* is used to implement the methods in exactly the same way that Oberon does. This mechanism is explained in detail in [RW92, Section 12.5], and we can summarize it by saying that the *handler* is a sort of *dispatcher* that receives a *message* and, depending on the *type* and the *content* of the message, performs the corresponding action.

In our persistent store, the handler needs to be externalized and in the current implementation, we decided to externalize the name of the module and the name of the procedure that constitutes the handler. Later, when the object is internalized, the system will first try to load the module (if it is not already loaded), then it will look for the procedure in the module and when it is found, it binds the handler to it. This means that the database does not only consists of the information stored in the persistent store, but also of all Oberon object files used by the data. The next step would consist in also storing the code in the persistent store, in order to make the store autonomous. With Oberon, the simplest way to do this would be to just copy the object-file in the store and to implement a special loader module that would load the code from the persistent store instead of an object-file. The whole system could even be fully portable if we use OMI's slim binaries [Fra94] instead of implementation specific object-files. We did not use OMI's slim binaries in our system because we would have had to rewrite a substantial part of the HP-Oberon compiler, and this was not in the scope of our work.

The last attribute of DB-Objects is the *belongsTo* attribute. This attribute is the key of our classification model and of our evolution mechanism. It is used to link the object with all collections to which it belongs. In this sense, this can be seen as the inverse of the *ext* attribute of the collections. In addition to representing the membership of a collection, the *belongsTo* attribute is also used to store the effective *values* of the attributes provided by that collection. Typically, an object belongs to several collections, and with our evolution model, the number of collections to which an object belongs can even change during the lifetime of an object. For these reasons, we need to use a dynamic structure to implement the *belongsTo* attribute, and we decided to implement it as a dynamic array. Here is the definition of the element type of this array:

```
Membership* = RECORD
  collection-: Object;
  attributes-: POINTER TO ARRAY OF Attribute
END
```

The *collection* attribute is a pointer to the collection to which the object belongs. The *attributes* attribute is used to store the values of the attributes provided by that collection. Once again the *attributes* field is implemented as a dynamic array. The different attributes provided by a collection do not necessarily have the same type, so the elements of this array are not necessarily homogeneous. With Oberon, it is possible to have such an heterogeneous structure by defining an empty, generic base type, and then, by extending this base type to another one that carries the information. Here is the definition of such a generic type:

```
Attribute* = POINTER TO AttributeDesc;
AttributeDesc* = RECORD
END;
```

And here is an extension of the previous type definition:

```
IntAttr* = POINTER TO IntAttrDesc;
IntAttrDesc* = RECORD(AttributeDesc)
  value*: LONGINT
END;
```

The *integer* attribute that we described is just an example and, using the same technique, we can also define *character* attributes, *string* attributes, etc. The fact that all these types have the same base type, namely *AttributeDesc* allows us to group them in a dynamic array.

Figure 9.2 summarizes the complex relationships between collections and objects. In this example, we see that object 42 is a member of the collections Persons and Professors. In the collection Persons, object 42 has an attribute name (John) and age (31). In the collection Professors, object 42 has the attribute uni (CMU). So, we can say that object 42 is a 31 year old person, that his name is John, and that he is professor at CMU.

9.3 Operations and Transactions

In the two preceding sections we have seen the structure used to represent our data model. Let us now explain how we implemented the operations for managing this structure.

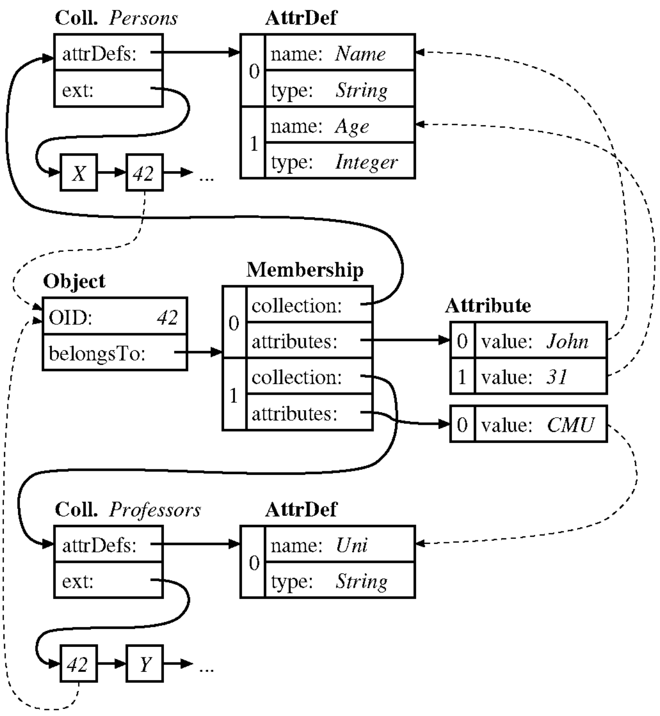

Figure 9.2: Implementation of Collections and Objects

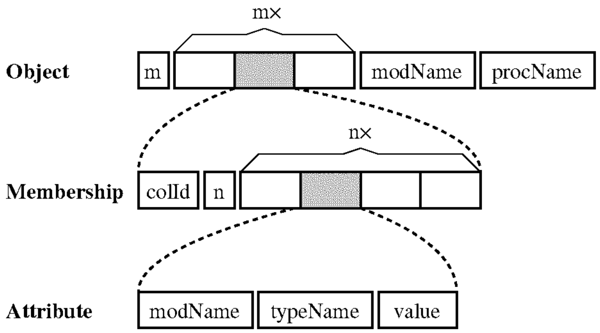

Figure 9.3: Serialized Representation of a DB-object

9.3.1 Loading and Storing DB-objects

The first operation that we want to describe is the loading and storing of DB-objects. In order to understand this process, we need first to show the serialized representation of a DB-object.

In Figure 9.3, we see that the object is represented by a list of *memberships*, followed by a *modName* and the *procName* that represents the *handler* of the object.

The *membership* is composed of a *colId*, representing the collection to which the object belongs, and a list of *attributes*.

The *attribute* itself is composed of a *modName* and the *typeName* that represents the *type* of the attribute, and a *value* that represents the actual value of the attribute. Actually, the information about the type of the attribute is redundant because, as we will see later, this information is also contained in the collection to which the object belongs. However, the internalization of an attribute will only work if its value corresponds with its type. Otherwise, the system may end in a totally inconsistent state. This is why we decided to add redundancy here.

Now that we know the structure of the serialized DB-object, it is easy to understand how it is loaded from and stored to the persistent store. Here are the pseudo-codes of the load operations:

```
PROCEDURE InternalizeObject (o: Object);
BEGIN
   ReadInt(m);
   IF m = 0 THEN o.belongsTo := NIL
```

```
    ELSE NEW(o.belongsTo, m) END;
    i := 0; WHILE i < m DO
      ReadLInt(id);
      o.belongsTo[i].collection := OldObject(id);
      ReadInt(n);
      IF n = 0 THEN o.belongsTo[i].attributes := NIL
      ELSE NEW(o.belongsTo[i].attributes, n)
      END;
      j := 0;
      WHILE j < n DO
        ReadString(modName); ReadString(typeName);
        M := Modules.ThisMod(modName);
        T := Types.This(M, typeName);
        Types.NewObj(o.belongsTo[i].attributes[j], T);
        o.belongsTo[i].attributes[j].Load(R);
        INC(j)
      END;
      INC(i)
    END;
    ReadString(modName);
    IF modName = "" THEN
      o.handler := NIL
    ELSE ObjectsReadString(procName);
      Refs.GetProcAdr(modName, procName, pc);
      o.handler := SYSTEM.VAL(ObjectHandler, pc);
    END
  END InternalizeObject;
```

As you can see in the preceding procedure, we are using the modules *Types* and *Refs* for accessing meta-information about the types and the procedures defined in a module. These two modules reflect the two main features of Oberon for accessing meta-informations.

The Type Descriptors. These descriptors are used by Oberon to support the Object Oriented Model of the language. They are used, among others, to check, at run time, the compatibility between two types. They are also used by the garbage collector to find all pointers contained in the objects.

The References. References are primarily used by the *Trap Viewer* to display the status of the Oberon system when an exception occurs. This meta-information is produced by the compiler and gives the name and the position in the object file of all variables and all procedures.

More information about meta-information in Oberon can be found in the work of J. Templ [Tem94].

The store operation is similar to the load, but instead of reading from the store, we just write to it. Here is the pseudo-code of this procedure:

```
PROCEDURE ExternalizeObject (o: Object);
BEGIN
   IF o.belongsTo = NIL THEN m := 0
   ELSE m := SHORT(LEN(o.belongsTo↑)) END;
   WriteInt(m);
   i := 0; WHILE i < m DO
      WriteLInt(o.belongsTo[i].collection.id);
      IF o.belongsTo[i].attributes = NIL THEN n := 0
      ELSE n := SHORT(LEN(o.belongsTo[i].attributes↑))
      END;
      WriteInt(n);
      j := 0;
      WHILE j < n DO
         attr := o.belongsTo[i].attributes[j];
         T := Types.TypeOf(attr);
         WriteString(T.module.name); WriteString(T.name);
         o.belongsTo[i].attributes[j].Store(R);
         INC(j)
      END;
      INC(i)
   END;
   IF o.handler = NIL THEN
      Streams.WriteString(R, "")
   ELSE pc := S.VAL(LONGINT, o.handler);
      Refs.GetProcName(pc, modName, procName);
      WriteString(modName); WriteString(procName)
   END
END ExternalizeObject;
```

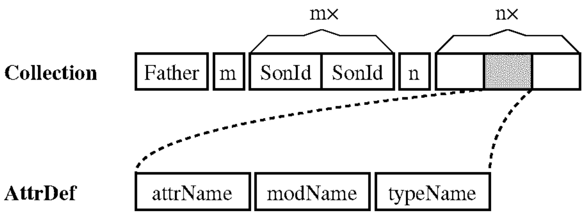

Figure 9.4: Serialized Representation of a collection

9.3.2 Loading and Storing collections

We already know that collections are DB-objects, so loading and storing collections begins with the loading and storing of the object parts of the collections in exactly the same way shown in the preceding sub-section. Then the process continues with the part that is specific to the collection. Here again, we first want to give the structure of the serialized collection.

In Figure 9.4 we see that a collection is described by an OID representing the father of a collection, by a list of OIDs representing the sons, and by a list of attribute definitions. Each attribute definition is then represented by the name of the attribute, the name of the module where the type of the attribute is defined and, finally, by the name of the attribute type.

We do not give the pseudo-code of the procedures here because they are implemented using the same mechanism previously used for objects.

9.3.3 Adding an object to a collection

Adding an object to a collection or updating the attributes of an object are probabely the most frequent changes in a database, so these operations have to be simple and efficient. In our system, updating the attributes of an object is a trivial and efficient operation because the system needs only to write the new object at the end of the store and to update its OID table. Let us now see how to add an object to a collection.

This operation consists of the following steps:

1. Check whether the collection already contains the object.
2. If the collection is a persistent one, then add the collection to the membership of the object.
3. Update the *ins* and the *del* fields.

4. Update the *ext* field.
5. Propagate the change to the father collection.
6. Tell the system that the collection has been updated by calling the *Touch* method.

Here is the process in pseudo-code:

```
PROCEDURE (c: Collection) Add (o: Objects.Object);
BEGIN
  IF (c # emptyCollection) & ~c.Has(o) THEN
    IF (c.ext IS Objects.PList) & (~o.HasCollection(c)) THEN
      o.AddCollection(c, LEN(c.attrDefs↑))
    END;
    IF c.del.This(o.id) # NIL THEN c.del.Delete(o)
    ELSE c.ins.Insert(o) END;
    c.ext.Insert(o);
    IF c.father # NIL THEN c.father.Add(o) END;
    c.Touch
  END
END Add
```

In the preceding code, we only show the simple propagation to the father of the collection. In fact, there are still other propagations, due to our constraints mechanism and to the indexes. We will treat the subject of update propagation in the next section.

9.3.4 Removing an object from a collection

Removing an object from a collection is slightly more complicated than adding an object to a collection. The problem is that an object could be referenced by another still existing object. For example an object to be deleted could be a member of the pair of an association, and removing only one side of the pair would make the association inconsistent. The following pseudo-code shows a simplified implementation of deletion.

```
PROCEDURE (c: Collection) Remove (o: Objects.Object);
BEGIN
  IF (c # emptyCollection) & c.Has(o) THEN
    IF o.HasCollection(c) THEN o.DropCollection(c) END;
    IF c.ins.This(o.id) # NIL THEN c.ins.Delete(o)
    ELSE c.del.Insert(o) END;
    c.ext.Delete(o);
    IF c.sons # NIL THEN
      i := 0; len := LEN(c.sons↑);
      WHILE i ¡ len DO
        x := Objects.OldObject(c.sons[i]);
        x.Remove(o); INC(i)
      END
    END;
    IF c.associations # NIL THEN
      i := 0; len := LEN(c.associations↑);
      WHILE i ¡ len DO
        a := c.associations[i](Association);
        IF a.lCol = c THEN
          pair := a.ThisLink(o, NIL);
          WHILE pair # NIL DO
            a.Remove(pair);
            pair := a.ThisLink(o, NIL)
          END
        ELSE
          ASSERT(a.rCol = c);
          pair := a.ThisLink(NIL, o);
          WHILE pair # NIL DO
            a.Remove(pair);
            pair := a.ThisLink(NIL, o)
          END
        END;
        INC(i)
      END
    END;
    c.Touch
  END
END Remove
```

Here again, we only show the simple propagation to the sons of the collection and to the associations. In the actual implementation, we also have the propagations due to the constraints mechanism and to the indexes. Actually, if we delete one member of a pair of an association, then the pair will have a dangling pointer and in order to resolve this inconsistency, we also have

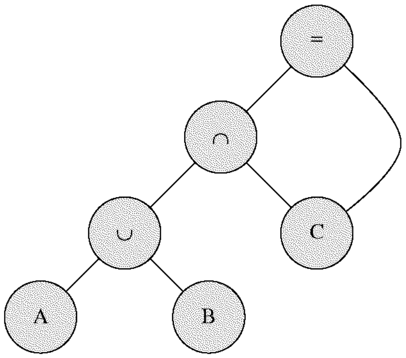

Figure 9.5: Tree representing the expression: $(a \cup b) \cap c = c$

to delete the pair in question from the association.

9.4 Constraints and Update Propagation

As we have written in Chapter 6, our system supports the concept of constraints to ensure the logical consistency of the data. These constraints are given in the form of an algebraic expression and are checked at the commitment of the transactions.

From the point of view of the implementation, constraints are represented by syntax trees that correspond to an algebraic expressions. As an example, Figure 9.5 shows the tree that corresponds to the constraint $(a \cup b) \cap c = c$ (which means that if an object is in collection c, then it has also to be in collection a or b)

At the end of a transaction, when the system has to check the constraints, it goes from the root of the tree and computes, for each node, the changes that have been made during the transaction. If the node is a collection (like a, b or c in our example), then these changes are simply given by the *ins* and *del* fields of the collection. However, if the node is an operation (like $\cup$ or $\cap$), then the system has to effectively compute these changes. When all nodes are computed, it is easy for the system to check if the expression evaluates to TRUE, i.e. whether the constraint is valid or not.

However, the interesting point here is not so much in the validation of the constraints, but rather in the *propagation* of updates. We have seen in chapter 6 that when our system encounters an inconsistency, it first tries to recover using update propagation. Let us see how this works: With the consistency constraint, the state of the database can only change when an object is inserted into a collection or removed from a collection.

As we have seen in chapter 6, the propagation mechanism consists of two phases, the *up* phase, where propagation is done from the leaves of the tree towards its root and the *down phase*, where propagation is done from the root to the leaves. The following procedure shows the implementation of the insert propagation:

```
PROCEDURE PropagateInsert(o: Objects.Object; n: ColNode);

   PROCEDURE Up(n: ConNode);
   BEGIN
      IF n IS ColNode THEN n(ColNode).collection.Add(o) END;
      IF n.father IS ConNode THEN
         father := n.father(OpNode);
         IF father.left = n THEN brother := father.right
         ELSE brother := father.left END;
         IF father.op = intersection THEN
            IF brother.Has(o) THEN Up(father) END
         ELSE
            IF ~brother.Has(o) THEN Up(father) END
         END
      ELSE (* Node is a Constraint *)
         con := n.father(Constraint);
         IF con.left = n THEN brother := con.right
         ELSE ASSERT (con.right = n); brother := con.left END;
         IF ~brother.Has(o) THEN Down(brother) END;
      END
   END Up;

   PROCEDURE Down(n: ConNode);
   BEGIN
      IF n IS ColNode THEN
         Up(n)
      ELSIF n(OpNode).op = intersection THEN
         Down(n(OpNode).left); Down(n(OpNode).right)
      END
   END Down;
```

```
BEGIN
  Up(n)
END PropagateInsert;
```

This procedure is an accurate representation of the algorithm that we have presented in section 6.5.

The delete propagation is very similar, so it is not necessary to give its code here as well. As we can see in the code, and thanks to our representation of consistency constraints, the propagation can be solved in a very easy and efficient way.

9.5 Object Evolution

In this section, we will show how easy it is to implement object evolution in our system. In fact, we do not need anything more than what has already been implemented, so let us show how we use the system to achieve object evolution.

Suppose that we have a collection of *students* and a collection of *assistants*, and that we want to make a student named “John” evolve from *students* to *assistants*. The *assistants* collection defines a string attribute called *department* where we store the department for which the assistant is working.

The first step is to locate “John” in the *students* collection. We will see in detail in the next chapter how such a query is done, but for the moment, let us suppose that we have a procedure *ThisStudent* that returns a student object given a name.

```
p := ThisStudent(”Jacques”);
```

Then we look for both collections *students* and *assistants* in the system root.

```
sc := RMS.ThisRoot(RMS.systemRoots, ”Students”);
students := sc(Collections.Collection);
ac := RMS.ThisRoot(RMS.systemRoots, ”Assistants”);
assistants := ac(Collections.Collection);
```

We add the person to the assistants collection:

```
assistants.Add(p);
```

We define the new attributes:

```
NEW(a); a.Assign(”Computer Science”);
assistants.SetObjectAttribute(p, ”department”, a);
```

We remove the person from the students collection:

```
students.Remove(p);
```

And we commit the transaction:

```
PSM.CommitTransaction
```

That’s it. Object evolution is quite straight-forward in our system. We do neither need special constructs nor special operations.

9.6 The Query Representation

Having described how the data is changed, how the information is updated and how it evolves we still have to explain how to search the information in the database using *queries*.

In our system, a query is represented by a binary syntax tree. Each node represents either a collection or an operation of the algebra shown in Chapter 5.

The base type of each node is *Item*, and it is defined as:

```
TYPE
   Item* = POINTER TO ItemDesc;
   ItemDesc = RECORD
      tag: INTEGER;
      s: Algebra.Selector;
      u, v: Item;
   END;
```

Such an item is able to represent all operations of our algebra. The field *tag* specifies the operation to be performed, the field *s* is used for the selection operations and the fields *u* and *v* denote the arguments of the operation.

To represent the objects of the query (here the collections), we define the following type:

```
Object = POINTER TO ObjectDesc;
ObjectDesc = RECORD (ItemDesc)
   collection: Collections.Container;
   next: Object
END;
```

In fact, this structure is also used to link all objects together (using the *next* field) and can be viewed as a symbol table. This table is much simpler than the one of a programming language like Pascal or Oberon because there is only one single type of symbol, denoting collections, and there is only a single scope.

In the current implementation, we do not have any query parser, so we still have to build the syntax tree of a query by writing an Oberon procedure. However, it is technically easy to add another type of front-end which would accept other forms of queries. Nowadays, the most common form a query is still represented by a text, expressed in accordance with the rules given by a *Query Language*. SQL [DD97] (Structured Query Language) is a widely used query language, but it has been designed for relational databases, and it does not fit well with our model.

Another possibility could be to implement the query language of OMS, namely AQL [NW99]. Our system is close enough to OMS to allow such a language. However, our research has gone towards the efficient persistence store and the data model and not towards the query language, and this is why we still have a primitive query mechanism.

Nevertheless, we want to show in this section how to use our system to build a query. Let us take the following example: "Give me the set of all professors in the computer science department who are also playing golf". Using our algebra (see Chapter 5), we can write this query in a formal way as:

$$(\text{Professors}\%\mathfrak{F}) \cap \text{GolfPlayers}$$

Where $\mathfrak{F}$ is an Oberon procedure that returns TRUE if the object passed as parameter is a member of the *Professors* collection and if, in this role,

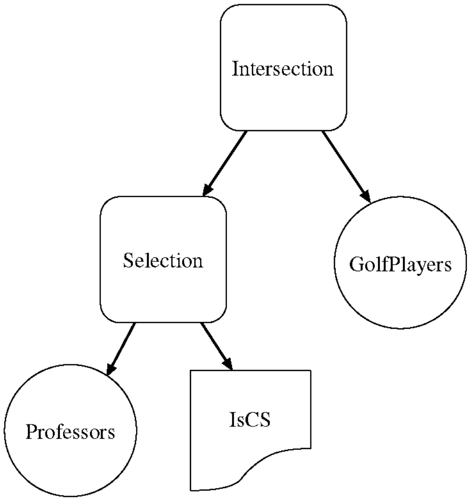

Figure 9.6: Example of a Query Syntax Tree

its *department* attribute is defined as “computer science”. Figure 9.6 shows the corresponding syntax tree, and the following procedure shows how the selector $\mathcal{F}$ is implemented.

```
PROCEDURE 𝓕 (o: Objects.Object;
   c: Collections.Container): BOOLEAN;
   VAR depAttr: Objects.Attribute;
BEGIN
   c(Collections.Collection).GetObjectAttribute(o,
      ”department”, depAttr);
   RETURN nameAttr(Objects.StringAttr).value↑
      = ”computer science”
END 𝓕;
```

9.7 The Query Evaluation

Once a query tree has been build, it can be passed to the *evaluator* which returns the collection resulting from the query expression. The implementation of the back-end is very simple; it just consists of traversing the syntax tree (in *postorder*), to look at the tag of each node, and to call the corresponding operation. The following code shows an excerpt of the *Evaluation* procedure.

```
PROCEDURE **Evaluation*** (x: Item): Collections.Container;
BEGIN
  IF x = EmptyCollection THEN
    RETURN Collections.emptyCollection;
  ELSE
    CASE x.tag OF
    | collection: RETURN x(Object).collection
    | union: RETURN Algebra.Union(Evaluation(x.u),
       Evaluation(x.v));
    | difference: RETURN Algebra.Intersection(Evaluation(x.u),
       Evaluation(x.v));
    | selection: RETURN Algebra.Selection(Evaluation(x.u),
       x.s);
    | dom: RETURN Algebra.Domain(Evaluation(x.u));
    | dr: RETURN Algebra.DomainRestriction(Evaluation(x.u),
       Evaluation(x.v));
    . . .
    END;
  END;
END Evaluation;
```

In module *Algebra* we find all operations defined in Chapter 5. In our implementation, all members of a collection are sorted by their OID and we can use this information to implement the operations more efficiently. For example, the following procedure shows how *Intersection* is implemented:

```
PROCEDURE **Intersection*** (x, y: Collections.Container):
  Collections.Container;
  VAR xi, yi: Collections.Iterator; z: TempCollection;
BEGIN
  z := NewDOpCollection(x, y);
  xi := x.NewIterator();
  yi := y.NewIterator();
  WHILE (xi.object # NIL) & (yi.object # NIL) DO
```

```
      WHILE (xi.object # NIL) & (xi.object.id < yi.object.id) DO
        xi.Step
      END;
      IF xi.object # NIL THEN
        WHILE (yi.object # NIL) & (xi.object.id < yi.object.id) DO
          yi.Step
        END
      END;
      IF (xi.object # NIL) & (xi.object # yi.object) THEN
        z.Add(xi.object); xi.Step; yi.Step
      END;
    END;
    RETURN z
  END Intersection;
```

As we can see, and thanks to the algebra defined in the first part, the implementation of a query mechanism was possible and even easy. We have also shown that our system can efficiently operate on collections, with one exception, namely *selection*. In fact, the problem with selection is that the procedure has to take each object of a collection and determine if it is accepted or not in the selection. This can be time consuming if the collections are huge. The standard solution to this problem is to use *indexes*, implemented as *B-Trees* or as *Hash-Tables*. We do not want to go into details of the implementation, but we just want to mention that our system also uses indexing mechanisms implemented as B-Trees to improve selection.

Chapter 10

Measurements and Performance Tests

Having implemented our system, we were interested in how big and how fast our database system is. It is difficult to compare this information with similar systems, because we do not have a similar system running on our workstation. However, we will try to compare ours with the OMS PRO system running on a Sun Sparcstation, and with the popular Postgress relational database system running on a PC.

One of the reasons why object-oriented database systems are not so popular is that they have the reputation of being slow and using large amount of volatile memory. Although this is true for some OODBMS, this does not necessarily have to be so. With these benchmarks, we want to show that an OODBMS can be fast enough to compete with RDBMS and that they can also be very economical in the use of memory.

The goal of this chapter is not to compare the features of the different databases, but to give an idea of the *speed* of our system. We want to show that our persistent store, custom-taylored, yields an attractive efficiency to our OODBMS.

10.1 Test Beds

The Odéon system runs on an HP-712 Workstation with a 66 MHz PA-RISC Processor and 64 MB of RAM. The Oberon System is the HP-Oberon system, started with a heap size of 16 MB. The disk on which the persistent objects are stored is a Fujitsu 778 MB SCSI Disk, rotating at 3'600 RPM.

The OMS system runs on Sun Ultra Sparc 16 MHz with 128 MB of RAM. It accesses its data on a remote disk using NFS (Networked File System) on a 10 MB/s network. This is certainly not the fastest way of accessing a database, however, it is not so much slower than our old Fujitsu disk. OMS has been written in Prolog using the SICStus database for its persistent store. We also have to say that OMS is a prototyping system and has not been optimized for dealing with a lot of objects.

The Postgress system runs on a PC with a 100 MHz Pentium Processor and 16 MB of RAM. The Operating System is Linux and the disk is a Western Digital EIDE disk, rotating at 5'200 RPM.

We know that, unfortunately, the systems are quite different, but this is the best we can do with the resources at our disposal.

10.2 Size of the System

Our OdéonSystem can be seen as consisting of five parts:

- The Oberon System running on Unix
- The Streams and Disk/IO Extensions
- The Persistent Store and the Persistent B-Trees
- The Objects and Collections Manager
- The Query system

Figure 10.1 shows the size of the object files in each part.

To summarize, we can say that

- the size of our system is about 152 KB and that it runs on a 258 KB big operating system.
- In comparison, the size of OMS is 716 KB and it uses a 1.4MB External Database Engine for its persistent store. The underlying Prolog system is 2.6 MB.
- On our machine, the Postgresql server is more than 6MB big.

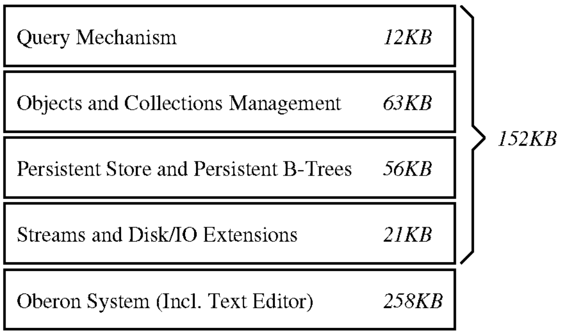
Query Mechanism 12KB
Objects and Collections Management 63KB
Persistent Store and Persistent B-Trees 56KB
Streams and Disk/IO Extensions 21KB
Oberon System (Incl. Text Editor) 258KB
152KB

Figure 10.1: The Size of the Object Files in Each Part

10.3 Speed of the System

In order to give an idea of the speed of our system, we decided to test the two most frequently used operations, namely the insertion of an object in a collection and the search of an object in a collection. We did not evaluate the speed of object evolution, because in our system, object evolution is equivalent to insertion. The efficiency of object evolution is thus equal to the speed of insertion of an object in a collection.

For these tests, we created a small database with two disjoint collections A and B. The disjoint constraint has to be automatically checked by the system.

In a first phase, we added 2’500 records in each collection, so we added a total of 5’000 records. On our HP, this operation was done in about 18 seconds and required about 1’900 disk read/write-accesses. We repeated this operation four times, ending with a database with 25’000 objects and the insertion of the last 5’000 objects was less than 3% slower than the first one and required 1% more disk access.

Then we repeated the same operation, but this time without constraint. Then the time needed for each insertion of 5’000 records was about 18 seconds with no noticable variance. This shows that our system is able to check simple constraints in a very efficient way. This is due to the implementation of our system, which can rapidly tell if a given object is a member of a collection or not. Another important point that leads to this efficiency is the fact that we took great care that as few objects as possible be checked.

After having measured the speed of insertion, we measured the speed of a simple query, namely the search of a given object in a collection. So we first filled a collection with 5'000 objects and started the query. The query was finished in about 800 msec with a total of 849 disk read-accesses. This good result was due to the fact that many objects where still in the caches of the system. So we flushed the caches and restarted the search. This time, it was done in 17 seconds. We were disappointed by this performance and used a profiler to see where all this time was spent. We found that about 30% of the time was spent in reading the meta information of the Oberon objects and 20% in accessing the disk. It is obvious that we could make our system much faster by improving the access to the meta information of Oberon, using a more efficient structure than the actual reference block [WG92, Page 438], usually used for debugging only.

In comparison, OMS needs 3 minutes and 52 seconds to insert the first 5'000 objects in its database and when the database already contains 20'000, adding 5'000 more objects takes 11 minutes and 16 seconds.

Postgresql needs more than 5 minutes to insert 5'000 records in a table and during this operation, we noticed that the CPU was only 10% loaded and that the operating system was not swapping, so we can conclude that most of the time was spent on disk operations and that Postgresql does not have an efficient caching mechanism. On the other hand, the retrieve operation was done in 1.55 seconds, which is a quite good result.

We can conclude this chapter by saying that our lean system is competitive and even faster than similar object-oriented database management systems. It is even competitive with *relational* database management systems like Postgresql. One could argue that Postgresql is not the fastest RDBMS on the market and that systems like MySQL are much faster, but unlike Postgress and our system, MySQL does not have any rollback facilities. This simplifies the storage mechanism significantly and opens the door to much better performance.

The good results shown in this chapter are primarily due to an efficient usage of the resources and good data structures.

One of the main arguments of the relational system defenders was that OODBMs were much slower than RDBMS; with our system, we hope to have demonstrated that this argument is no longer valid.

Chapter 11

Summary, Conclusion and Future Works

With this dissertation, we have presented the concepts of Odéon, an object-oriented database system based on the *OM* model and we have given a detailed description of its implementation. With this system, we have shown two essential points:

- It is possible to implement a simple but yet efficient Object Oriented Database Management System in Oberon.
- Using the idea of "Typing by classification", it is possible and even easy to solve the problem of object evolution.

The first point, namely the efficiency of the system, is primarily due to the implementation of the persistent store using the concept of the *log structure*. The second contribution to the efficiency is certainly a good usage of the available resources – disk and memory. Thanks to these techniques, our system is much faster than similar OODBMs and even competes well with existing relational systems.

It was possible to realize this persistent store with Oberon, because Oberon has very good support for object oriented design and is a very open system. Oberon allowed us to access the disk directly, that is, without the overhead of a file system. The only change we had to do to the Oberon system was to add the concept of *finalizer*. But this was not difficult because it had already been done in some other implementations of the Oberon System. Oberon is a small but efficient operating system, and so is our database management system Odéon.

The second fundamental aspect of our work, namely the way we solved the problem of object evolution is characterized by the idea of "Typing by classification". In traditional models, objects can only have one single type; in object-oriented models, objects still have one type, but they are *compatible* with all its *supertypes*. In our model, we deviate from this view and allow an object to be of several types. Our idea was to be closer to reality, where a given object may be viewed from different angles, or through different contexts, each revealing only the attributes specific to that context. In many applications, as in real life, we do not have only one object that can be seen through a given context, but a whole *collection* of objects.

Therefore, in our system – as in the *OM* data model, we decided to use the collections to group objects that have common attributes. These collections are the support for the classification of objects. The novelty of our system was to also use this same classification to define the *type* of the collection members. In other words, if an object is a member of a collection, it then has all the attributes defined by the collection. We called this technique *Typing by Classification* and as we have seen in this dissertation, this technique enables straightforward object evolution, and this in a very efficient way.

As OMS Java, our system could be extended in many different ways, and is thus an interesting platform to experiment with object-oriented database systems. Here is a list of possible extensions:

- Implement a Query Language
- Extend the transaction mechanism to allow multiple simultaneous access to the database
- Experiment with the garbage collector of the persistent store.
- Extend the persistent store to a distributed storage system

The first extension would be the design and implementation of a query language for Odéon. This language could be based on AQL and will allow to use the power of the algebra in a simplified way. This language will also allow interactive queries and make the system more attractive.

The second extension goes toward a system available for many simultaneous users. For the moment, only one user can access the database at a given time and this could be improved by a better transaction mechanism. However, such transaction mechanisms are not trivial and the effort needed to implement them may be considerable. But, we think that this extension is needed if we want to use Odéon for serious applications.

Another improvement of the system could be in the garbage collector of the persistent store. In the current implementation, the garbage collector works well, but sometimes, it has still to lock the database to do its work. It is possible to improve the collaboration between the system and the garbage collector to improve the availability of the system.

A last extension would be to distribute the data on different locations. This could be different disks in the same computer, but this can also be different computers connected by a network. The challenge in such systems is to limit the traffic on slow connections (like wide area network) and to favour fast ones.

Odéon is a small and efficient database management system, it implements the rich data-model *OM* and has an efficient object evolution mechanism. Odéon and Oberon are both open and easily extensible. This can lead to many interesting further developments.

Appendix A

User Manual and Tutorial

The goal of the tutorial is to give to the user an idea on how the Odéonsystem can be used to implement a small database. This database should contain information for equestrian sport, so we want to store information about horses, riders and their trainers. The Figure A.1 shows a graphical representation of the schema of our database.

In this schema, we will also add a constraint to avoid an object finding itself in both collections *Persons* and *Horses*.

A.1 Creating the Database

The first operation is the creation of the database from the schema that we showed just before. In other words, we have to create the collections first, define their attributes and create the association and the constraint. All these operations are grouped in the *CreateDatabase* procedure.

```
PROCEDURE CreateDatabase*;
   VAR persons, riders, trainers, horses: Collections.Collection;
   rides: Collections.Association; con: Collections.Constraint;
BEGIN
   (* Create a new collection, without super collection and
   . . . with 1 attribute *)
   persons := Collections.NewCollection(NIL, 1);
   (* Define the first attribute *)
   persons.SetCollectionAttribute(0, ”name”,
```

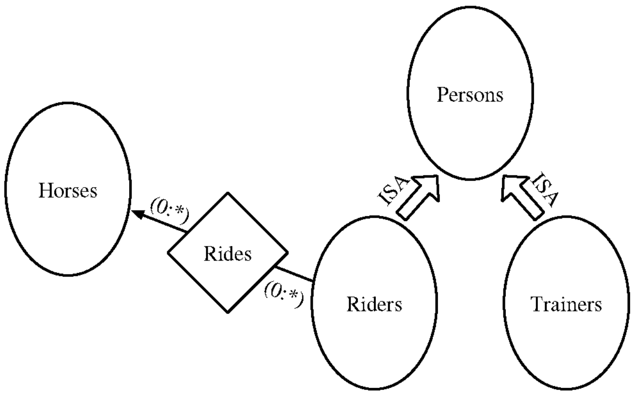

Figure A.1: The Equestrian Database Schema

```
   Objects.StringAttrType);
(* Store the collection c in the system roots under the
... name “Equestrian.Persons” *)
RMS.AddRoot(persons, ”Equestrian.Persons”);

(* Create a new sub-collection “Equestrian.Riders” of Persons
... with 1 additional attribute *)
riders := Collections.NewCollection(persons, 1);
riders.SetCollectionAttribute(0, ”license”,
   Objects.StringAttrType);
RMS.AddRoot(riders, ”Equestrian.Riders”);

(* Create the sub-collection “Equestrian.Trainers” of
... Persons with 1 additional attribute *)
trainers := Collections.NewCollection(persons, 1);
trainers.SetCollectionAttribute(0, ”club”,
   Objects.StringAttrType);
RMS.AddRoot(trainers, ”Equestrian.Trainers”);

(* Create the collection “Equestrian.Horses”, without
... super collection and with 1 attribute *)
horses := Collections.NewCollection(NIL, 1);
horses.SetCollectionAttribute(0, ”name”,
      Objects.StringAttrType);
    RMS.AddRoot(horses, ”Equestrian.Horses”);

    (* Create the association “Equestrian.Rides” between
    . . . riders and horses *)
    rides := Collections.NewAssociation(riders, horses, NIL, 0,
      0, Inf, 0, Inf);
    RMS.AddRoot(rides, ”Equestrian.Rides”);

    (* Set a constraint to ensure that horses and persons are
    . . . disjoint collections *)
    con := Collections.MakeConstraint(
      Collections.MakeOpNode(
        Collections.intersection,
        Collections.MakeColNode(horses),
        Collections.MakeColNode(riders)),
      Collections.MakeColNode(Collections.emptyCollection));
    PSM.CommitTransaction
  END CreateDatabase;
```

A.2 Inserting Objects

After having created the collections, we can start inserting objects into them. The first procedure shows how we can create *Persons* objects and how to insert them in the *Persons* collection.

```
  PROCEDURE AddPerson(name: ARRAY OF CHAR);
    VAR c: Collections.Collection; o: PSM.Object;
      p: Objects.Object; a: Objects.StringAttr;
  BEGIN
    (* Find the collection in the system roots *)
    o := RMS.ThisRoot(”Equestrian.Persons”);
    IF o # NIL THEN
      c := o(Collections.Collection);
      (* Create the person object and insert it into the collection *)
      p := Objects.NewObject(); c.Add(p);
      (* Create the attribute and assign its value *)
      NEW(a); a.Assign(name);
      (* Set the attribute of the object in the context of
      . . . the collection *)
      c.SetObjectAttribute(p, ”name”, a)
    ELSE
      ShowError(”I can’t find the collection Equestrian.Persons”)
    END
  END AddPerson;
```

The second procedure shows how to insert *Riders*. Remember the *Riders* collection is a sub-collection of *Persons*, so by adding an object to the *Riders* collections we also add it automatically to the *Persons* collection and it also gets the attributes of *Persons*.

```
  PROCEDURE AddRider(name, license: ARRAY OF CHAR);
    VAR c: Collections.Collection; o: PSM.Object;
      p: Objects.Object; a: Objects.StringAttr;
  BEGIN
    o := RMS.ThisRoot(”Equestrian.Riders”);
    IF o # NIL THEN
      c := o(Collections.Collection);
      p := Objects.NewObject(); c.Add(p);
      NEW(a); a.Assign(name);
      c.SetObjectAttribute(p, ”name”, a);
      NEW(a); a.Assign(license);
      c.SetObjectAttribute(p, ”license”, a)
    ELSE
      ShowError(”I can’t find the collection Equestrian.Riders”)
    END
  END AddRider;
```

We have only shown the procedures to insert objects in a collection, but it is easy to now implement commands that use these procedures or to call them from a GUI.

A.3 Searching Objects

Now that the collections contain data, we can show how to make a query. In this example, we will show a simple *Selection* for finding a horse by its name. If the horse does not exist or if more than one horses have the name by which we are searching, the procedure should return NIL. Here is the procedure:

```
PROCEDURE ThisHorse(name: ARRAY OF CHAR):
   Objects.Object;
   VAR o, x: PSM.Object; c: Collections.Container;
BEGIN
   COPY(name, thisHorse);
   o := RMS.ThisRoot(”Equestrian.Horses”);
   c := Algebra.Selection(o(Collections.Collection), HorseSel);
   (* Check that the collection contains only one object *)
   IF (c # NIL) & (c.Cardinality() = 1) THEN
      x := Collections.Singleton(c);
      RETURN x(Objects.Object)
   ELSE
      RETURN NIL
   END
END ThisHorse;
```

This procedure needs a *selector* for doing its task. Here is how the selector is implemented:

```
PROCEDURE HorseSel (o: Objects.Object;
   c: Collections.Container): BOOLEAN;
   VAR nameAttr: Objects.Attribute;
BEGIN
   c(Collections.Collection).GetObjectAttribute(o, ”name”,
      nameAttr);
   RETURN nameAttr(Objects.StringAttr).value↑ = thisHorse
END HorseSel;
```

We cannot add additional parameters to the selector, so we have to use a global variable to store the name of the horse that we are looking for:

```
VAR thisHorse: ARRAY 256 OF CHAR;
```

A.4 Evolution

The last operation that we want to show in this tutorial is the evolution. To illustrate this operation, we want to take a rider and promote him to a trainer. As you can see, the operation is very simple:

```
PROCEDURE UpdateRiderToTrainer* (name,
   club: ARRAY OF CHAR);
   VAR o: PSM.Object; p: Objects.Object;
      trainers, riders: Collections.Collection; a: Objects.StringAttr;
BEGIN
   p := ThisRider(name);
   o := RMS.ThisRoot(”Equestrian.Riders”);
   riders := o(Collections.Collection);
   o := RMS.ThisRoot(”Equestrian.Trainers”);
   trainers := o(Collections.Collection);
   trainers.Add(p);
   NEW(a); a.Assign(club);
   trainers.SetObjectAttribute(p, ”club”, a);
   riders.Remove(p);
   PSM.CommitTransaction
END Update;
```

In this short tutorial we have shown how an Oberon program could use the Odéonsystem. We have just shown the *essence* of the application, and a commercial application must also provide more input checks and probably also a graphical user interface, but these aspects are not in the context of this chapter.

Bibliography

[ABC$^+$83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. Ps-algol: A language for persistent programming. In *Proceeding of the 10th Australian National Computer Conference*, pages 70–79, Melbourne, Australia, 1983. 2.1

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceeding of the 19th VLDB Conference*, pages 39–51, Dublin, Ireland, 1993. Morgan Kaufmann. 2.2, 3.3

[ACC81] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-algol: an Algol with a persistent heap. *ACM SIGPLAN Notice*, 17(7), July 1981. 3.6

[Atk89] M. C. Atkins. *Implementation Techniques for Object-Oriented Systems*. PhD thesis, University of York, UK, June 1989. 7.6.4

[Bay72] R. Bayer. Symetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972. 7.6.3

[BG88] Dina Bitton and Jim Gray. Disk Shadowing. In *14. Intl. Conf. on Very Large Data Bases*, pages 331–338, Los Angeles, California, USA, August 1988. Morgan Kaufmann. 7.1

[BM72] R. Bayer and E. McCreight. Organisation and maintenance of large ordered indexed. *Acta Informatica*, (1):173–189, 1972. 7.9

[BMR95] Frank Buschmann, Regine Meunier, and Hans Rohnert. *Pattern-Oriented Software Architecture*. John Wiley & Sons, Incorporated, 1995. 1.1

[CLG$^+$94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994. 7.1

[Com79] Douglas Commer. The ubiquitous b-tree. *Computing Surveys*, 11(2):121–137, June 1979. 7.9

[DD97] C.J. Date and Hugh Darwen. *A Guide to the SQL Standard*. Addison Wesley, fourth edition, 1997. 5.2.1, 9.6

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, second edition, 1994. 3.3, 3.4, 3.5, 6.4

[FM98] André Fischer and Hannes Marais. *The Oberon Companion*. vdf Hochschulverlag AG an der ETH Zürich, 1998. 2.1

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996. 1.1

[Fra94] Michael Franz. *Code Generation On the Fly: A Key for Portable Software*. PhD thesis, ETH Zürich, Switzerland, 1994. 9.2

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, 1992. 8

[GSR96] G. Gottlob, M. Schrefl, and B. Röcki. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), July 1996. 2.2, 3.3

[GVJH94] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994. 1.1

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., 1990. 7.4

[Kna96] Markus Knasmüller. Adding persistence to the Oberon-System. Technical Report 6, Institut für Informatik, Johannes Kepler Universität Linz, Austria, 1996. 2.1

[KNW98] A. Kobler, M. C. Norrie, and A. Würgler. Oms approach to database development through rapid prototyping. In *Proc. of 8th Workshop on Information Technologies and Systems*, Helsinki, Finland, 1998. 2.2

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989. 2.1

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988. 1.1

[MMS79] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical report, Xerox Palo Alto Research Center, USA, 1979. 7.3

[Mös95] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, second edition, 1995. 3.4

[Nor92] Moira C. Norrie. *A Collection Model for Data Management in Object-Oriented Systems*. PhD thesis, University of Glasgow, Scotland, 1992. 3, 3.5, 5.2

[Nor93] Moira C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *12th Intl. Conf. on Entity-Relationship Approach*, pages 390–401, Dallas, Texas, December 1993. Springer-Verlag, LNCS 823. 2.2

[NW98] M. C. Norrie and A. Würgler. Oms rapid prototyping system for the development of object-oriented database application systems. In *Proc. Intl. Conf. on Information Systems Analysis and Synthesis*, Orlando, USA, 1998. 2.2

[NW99] M. C. Norrie and A. Würgler. Oms pro: Introductory tutorial. Technical report, Institut für Informationssysteme, ETH Zürich, March 1999. 2.2, 9.6

[OD89] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, January 1989. Also appears as University of California, Berkeley, Technical Report UCB/CSD 88/467. 7.2

[RDH$^+$80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980. 7.3

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992. 7.2

[RT74] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974. 7.3

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon. Steps beyond Pascal and Modula*. Addison Wesley, 1992. 1.4, 3.1, 7.5.3, 9.2

[SKN98] A. Steiner, A. Kobler, and M.C. Norrie. Oms/java: Model extensibility of oodbms for advanced application domains. In *Proc. 10th Conf. on Advanced Information Systems Engineering*, Pisa, Italy, 1998. 2.2

[SM77] Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science*. Prentice Hall International, 1977. 3.2

[SM80] Joachim W. Schmidt and Manuel Mall. Pascal/r report. Technical Report 66, Universitaet Hamburg, Germany, 1980. 2.1

[SS72] J. E. Stoy and C. Strachey. OS6 - an experimental operating system for a small computer. part 2: Input/output and filing system. *The Computer Journal*, 15(3):195–203, August 1972. 7.3

[Sup94] Jacques Supcik. HP-Oberon™: The Oberon implementation for Hewlett-Packard Apollo 9000 Series 700. Technical Report 212, Institute for Computer Systems, ETH Zürich, Switzerland, February 1994. 1.4

[Szy92] Clemens A. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems*. PhD thesis, ETH Zürich, Switzerland, 1992. 7.5.3, 7.6.4

[Szy98] Clemens Szyperski. *Component Oriented Programming: Beyond Object Oriented.* Addison-Wesley, 1998. 1.1

[Tem94] Josef Templ. *Metaprogramming in Oberon.* PhD thesis, ETH Zürich, Switzerland, 1994. 7.5.3, 7.6.4, 9.3.1

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon.* Addison Wesley, 1992. 10.3

[Wir86] Niklaus Wirth. *Algorithmen und Datenstruktur mit Modula-2.* B.G. Teubner Stuttgart, 1986. 7.6.3

Curriculum Vitæ

	Jacques Supcik
August 30, 1967	born in Fribourg citizen of Fribourg, FR son of Denise and Pierre Supcik
1974-1980	primary school in “la Vignettaz”, Fribourg
1980-1983	secondary school in “Jolimont”, Fribourg
1983-1987	Collège St-Michel in Fribourg
1987	Maturité (type C)
1987-1992	studies in computer science at the Swiss Federal Institute of Technology (ETH), Zürich
1992	Diploma (Dipl. Informatik-Ing. ETH)
1993-1998	research and teaching assistant at the Institute for Computer Systems, Swiss Federal Institute of Technology (ETH), Zürich in the research group headed by Prof. Dr. N. Wirth